# Deep Learning

## Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2025-08 Dozent: Karsten Flügge

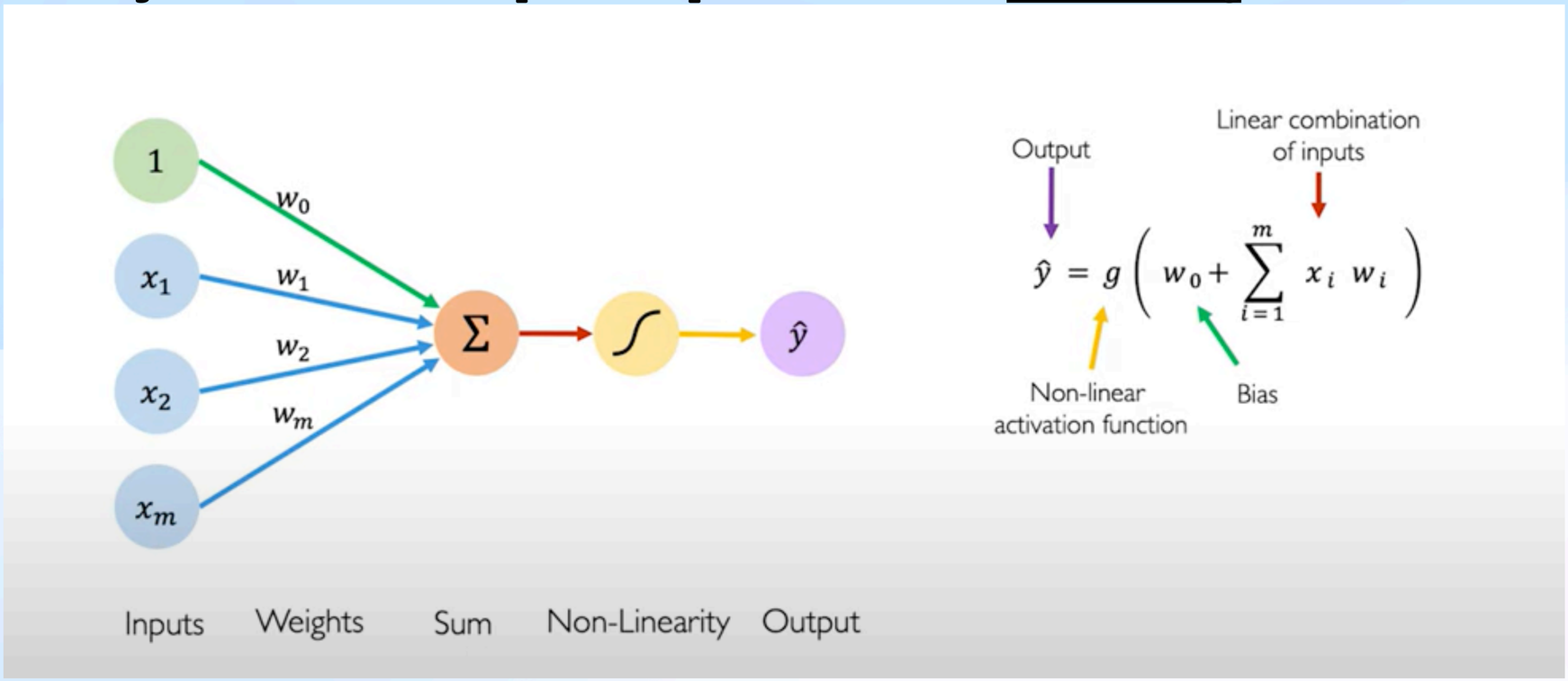
# Themen des Tages

## Tag 2

Perceptron
Parameter,
Fehler,
Optimierung
Rechenschemen: Backpropagation

# One Layer Perceptron

**One Layer Perceptron  $\bar{y} = g( W*x + w_0 )$ ($w_0$ constant bias)**
**Die Werte der Matrix W werden als Gewichte $w_0 \dots w_n$ bezeichnet**
**Als Aktivierungsfunktion / Nicht-Linearität g wird oft ReLU, TanH, Sigmoid oder 'Step' verwendet.**
**Das Ergebnis eines Perceptron Layers wird als "Aktivierung" bezeichnet**




**Um die Aktivierung $\bar{y}$ vom gewünschten output y (training-target) zu unterscheiden schreibt man oft einen Strich / Hut drüber, Elemente-weise $\bar{y}_i$ vs $y_i$ (y true labels vs y predicted)**

# One Layer Perceptron

Idee: reagiere auf verschiedene Eingaben (input x) mit verschiedener Ausgabe (output y)

Dazu wird jedem Teil der Eingabe (Element im Vektor) ein Gewicht zugeordnet und multipliziert:

x1*w1  w1 gross => x1 ist wichtig
x1*w1  w1 klein => x1 ist unwichtig

diese gewichteten Elemente werden dann zusammen addiert:

Summe = x1*w1 + x2*w2 + x3*w3 …   "skalarprodukt x°w"

Um etwas mehr Flexibilität zu haben gibt man noch ein Gewicht unabhängig vom input hinzu:

Summe = x1*w1 + x2*w2 + x3*w3 … + 1*w0

Das ist der sogenannte Bias w0

**Zusammen $\sum x_n w_n$  mit $x_0=1$**
**"Sammle die Eingaben und gewichte sie"**

# One Layer Perceptron

Idee: reagiere auf verschiedene Eingaben (input) mit verschiedener Ausgabe (output)

**Die Summe $\sum x_n w_n$ nennt sich "lineare" Schicht.**

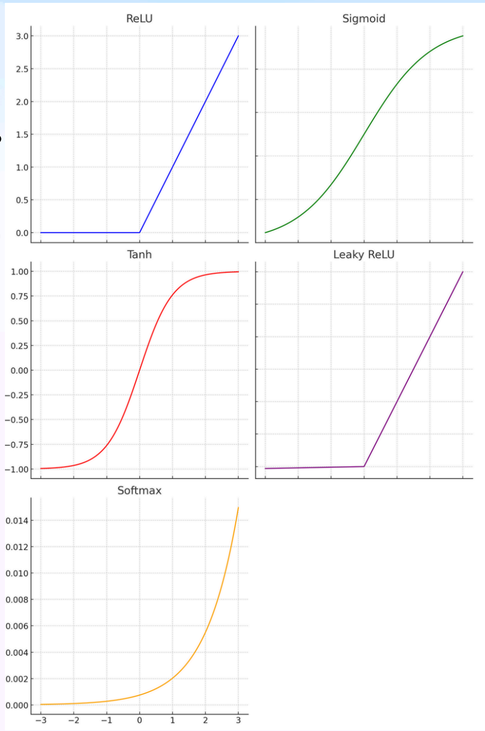
Das kommt daher dass jeder Parameter $x_n w_n$ wie eine lineare Gerade ist.

Typischerweise wird nach der Aufsummierung noch geschaut, ob das Ergebnis über einen gewissen Schwellwert kommt, also ob das Perceptron 'feuern' soll oder nicht, beziehungsweise wie stark.

Dies geschieht mit der sogenannten Nichtlinearität oder Aktivierungsfunktion.

Nicht-linear heisst, dass die Aktivierungsfunktion einen Knick hat oder krumm ist.

💡 mit einer Geraden allein lässt sich kein Schwellwert modellieren



# One Layer Perceptron

**Idee: reagiere auf verschiedene Eingaben (input) mit verschiedener Ausgabe (output)**

**Die Summe $\sum x_n w_n$ nennt sich "lineare" Schicht.**

**Wenn man den input x und die Gewichte w als Vektor betrachtet dann gibt es eine andere mathematische Schreibweise hierfür:**

$x \circ w = \sum x_n w_n$ Skalarprodukt "dot-product"

# One Layer Perceptron

$w \circ x = \sum w_n x_n$  Skalarprodukt "dot-product" 1-d output

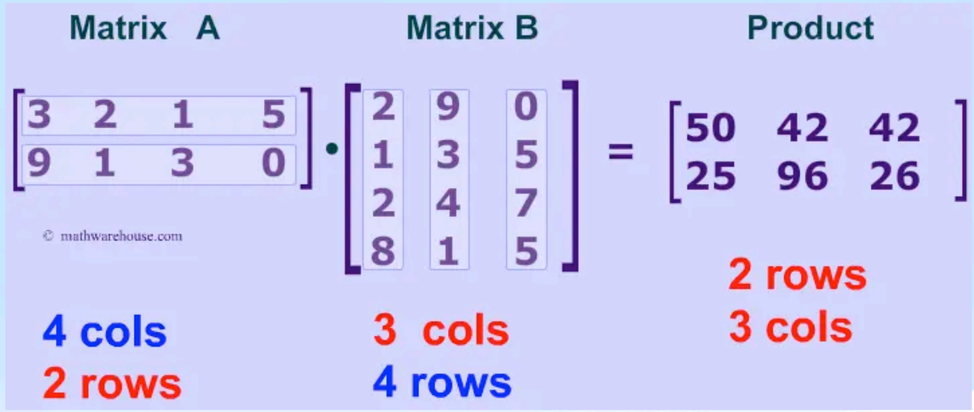
in python:

```
import numpy as np

# Example vectors
x = np.array([1, 2, 3]) # x[0] ist 1 und gehört zum Bias w[0]
w = np.array([4, 5, 6])

# Scalar product
dot_product = np.dot(x, w)
print(dot_product)
g = np.tanh # or np.sigmoid Activation function
y = g(dot_product)
```

# One Layer Perceptron als Matrix multiplikation

$W*x = \sum w_{n\,i} x_n$  Skalarprodukt "matrix-product" n-d output

```
# Example vectors
W = np.array([[4, 5, 6], [7, 8, 9]])
x = np.array([1, 2, 3])

# Matrix product
# W*x = [w1∘x, w2∘x] in numpy : W@x

# Scalar product
matrix_product = np.matmul(W, x) # or [np.dot(w1, x), np.dot(w2, x)]
print(matrix_product)

g = np.tanh # or np.sigmoid Activation function
y = g(matrix_product) # broadcast "erweitert Aktivierungsfunktion auf alle Elemente"
print(y)
```



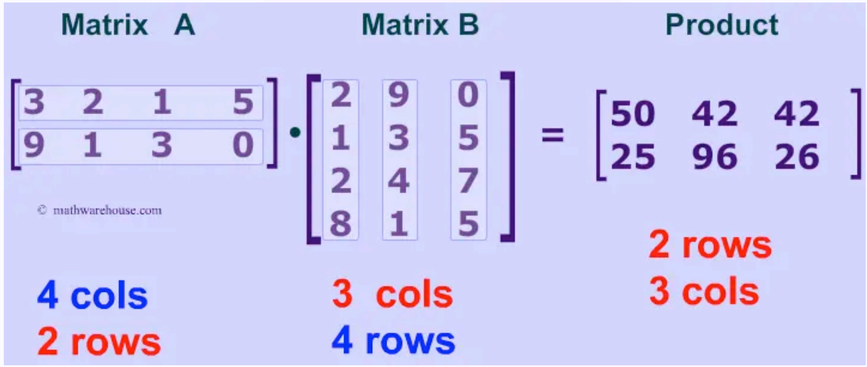
# One Layer Perceptron als Matrix multiplikation

Gruppierung verschiedener Gewichtungen

$w*x = \sum w_n x_n$ Skalarprodukt Summe von input **x** und Gewichten **w**
**w für weights**

andere $w_2$ **für weights für andere Aspekte**
$\mathbf{w*x = \sum w_n x_n}$

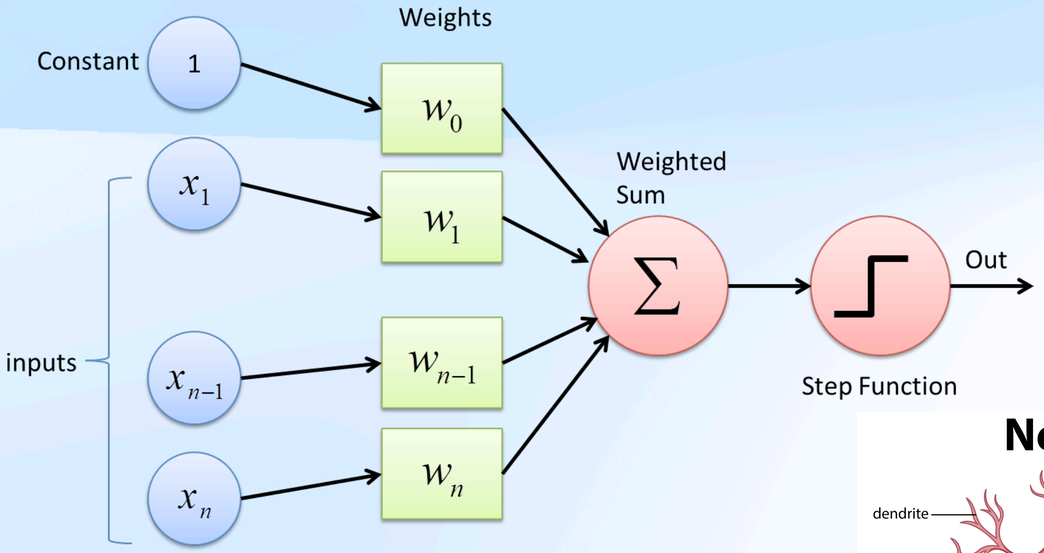
$\mathbf{W}*x = \sum w_{n\,i} x_n$ Skalarprodukt "matrix-product" n-d output

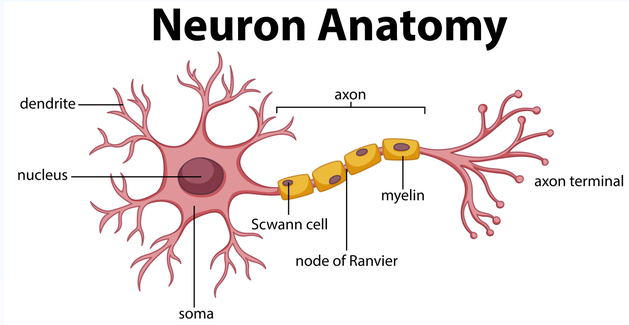


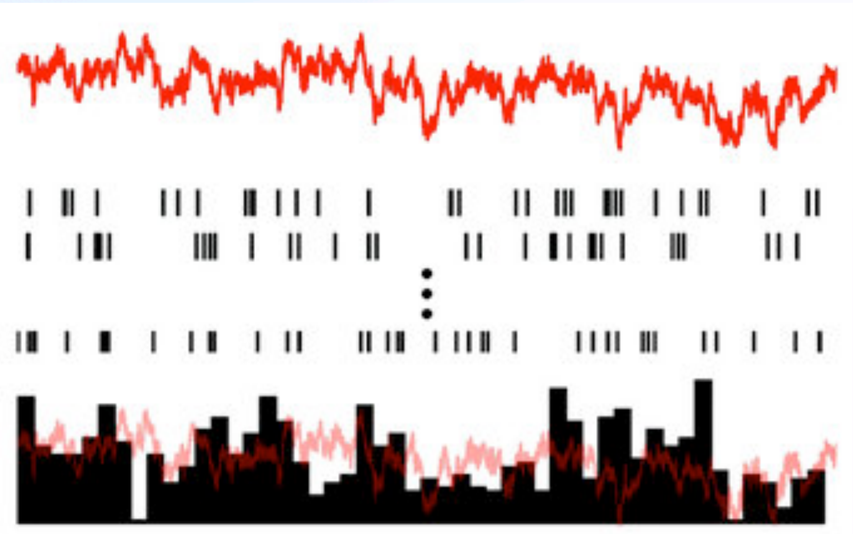
# One Layer Perceptron

**One Layer Perceptron  y = g( M*x + w0 ) (w0 constant bias)**
perceive "wahrnehmen, aufnehmen" *per* (“by, through”) + *capiō* (“to take”)

**Analogie zu Neuron: Die Gewichte verbinden andere Ausgänge (Axone) und werden ähnlich in *Synapsen* zusammenaddiert. Die "Aktivierungsfunktion" entspricht ungefähr den binären Spikes von Neuronen. Ein Neuron allein ist natürlich viel komplizierter, aber im Ensemble kann unser Perceptron vielleicht Grundfunktionen vom Neuron abbilden.**



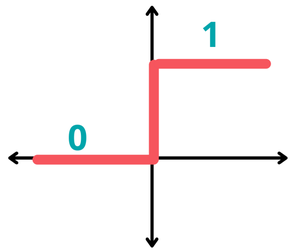
# Aktivierungen

**Sinn von Aktivierungsfunktionen:**

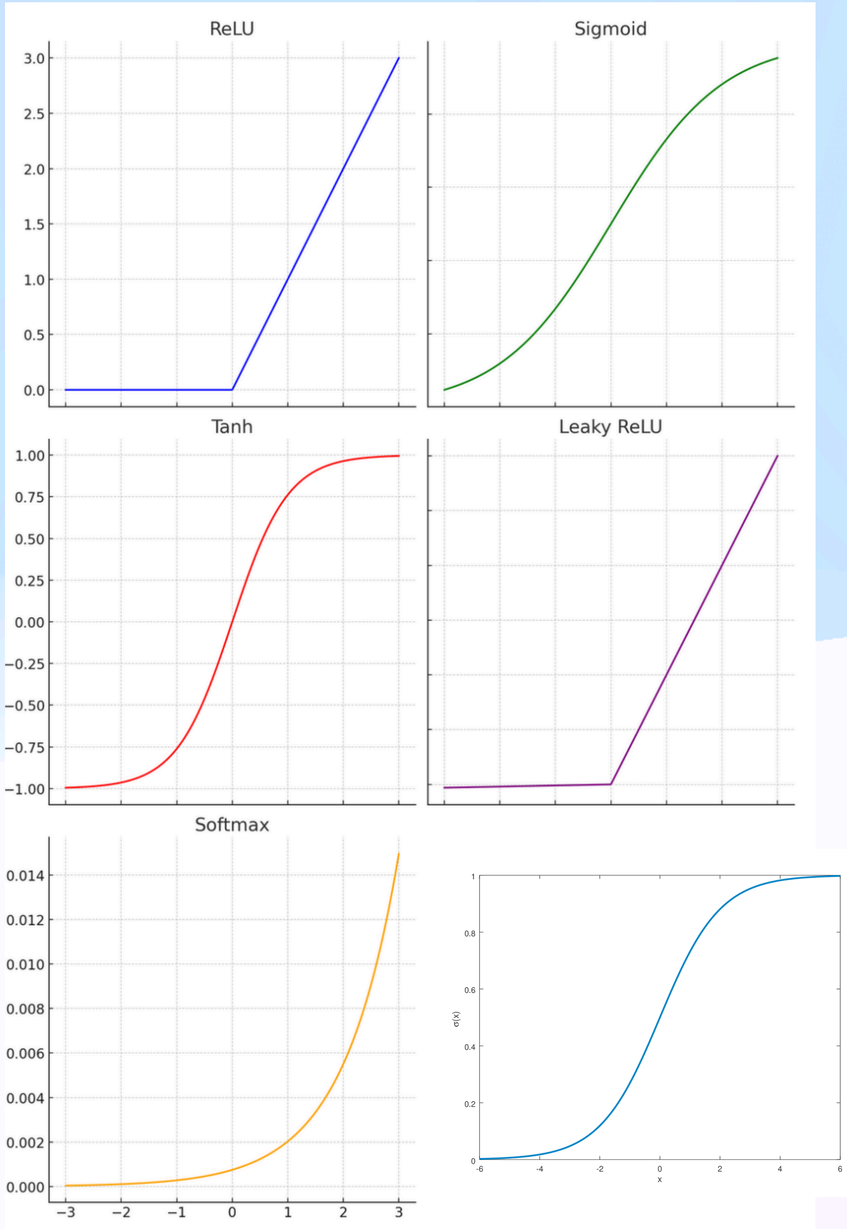
- **Unser "Neuron" "feuert" wenn ein Schwellwert erreicht ist**
- **(Optional) "Neuron" kann "gesättigt" sein (sigmoid/tanh)**

**Relu(x)= max(0,x)  Rectified Linear Unit**
**LeakyRelu(x)= max(εx,x) ε≈0  besser**
**Step(x)= x > 0**
**Sigmoid(x) = 1 / (1 + e⁻ˣ)**
**TanH(x)**

**Warum ist Step eine schlechte Aktivierungs-Funktion?**
**step(x) = x>0 ? 0 für x<0 1 für x>0**




Binary Step Activation Function



# Synonyme

**Perceptron**
**Linear Layer**
**Fully Connected Layer** (vs sparse layer mit vielen 0en)

**MLP multi layer perceptron**

# 2*Linear

**Warum machen zwei lineare Schichten ohne Aktivierung hintereinander keinen Sinn?**

**"Linear und linear ist wieder linear"**

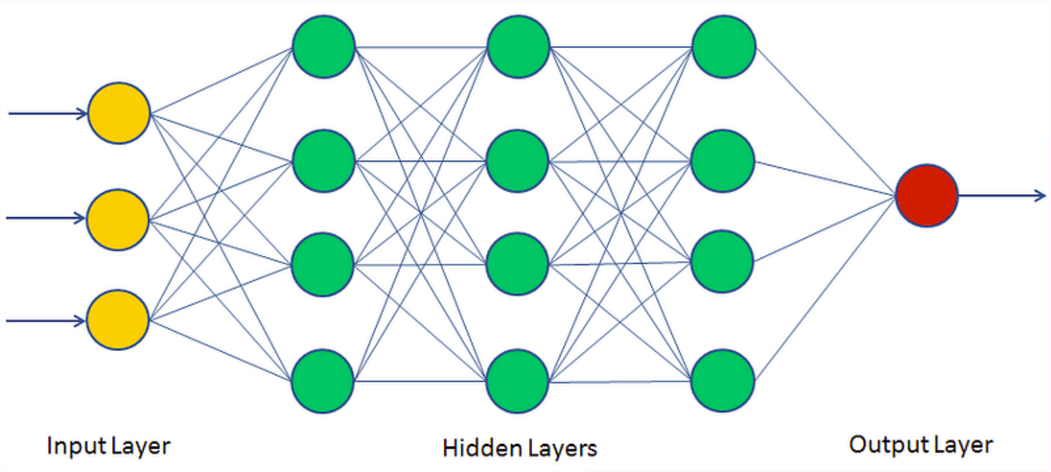
**f(x)=a*x**

**g(z)=b*z**

**g(f(x))=b*f(x) = b*a*x**

**dann kann man auch direct Schreiben a*b = c**

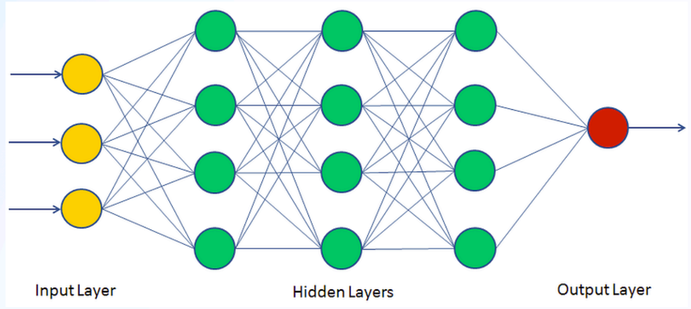
**g.f(x) = c*x**

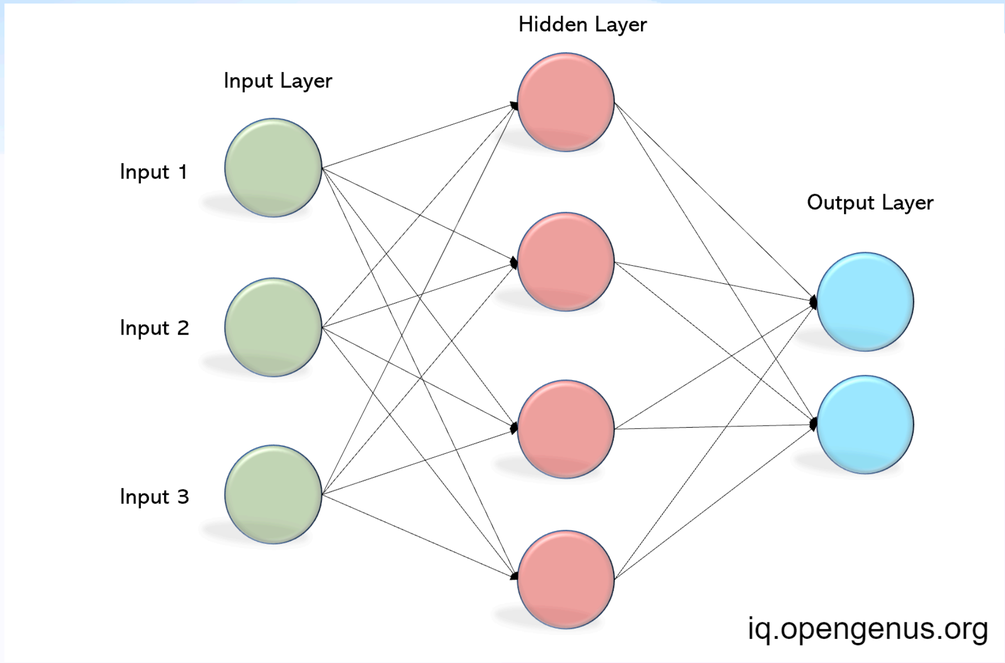


# Multi Layer Perceptron

**Wenn wir unser Input Vektor x mit Gewichten W und Aktivierung zu output Vektor y umrechnen, kann man noch ein Perceptron 'hinterschalten'.**
**Damit hat man mehrere 'Schichten'.**

**Die mittleren Schichten**
**nennt man 'hidden' (versteckt)**
**weil sie beim output**
**nicht zu sehen sind.**



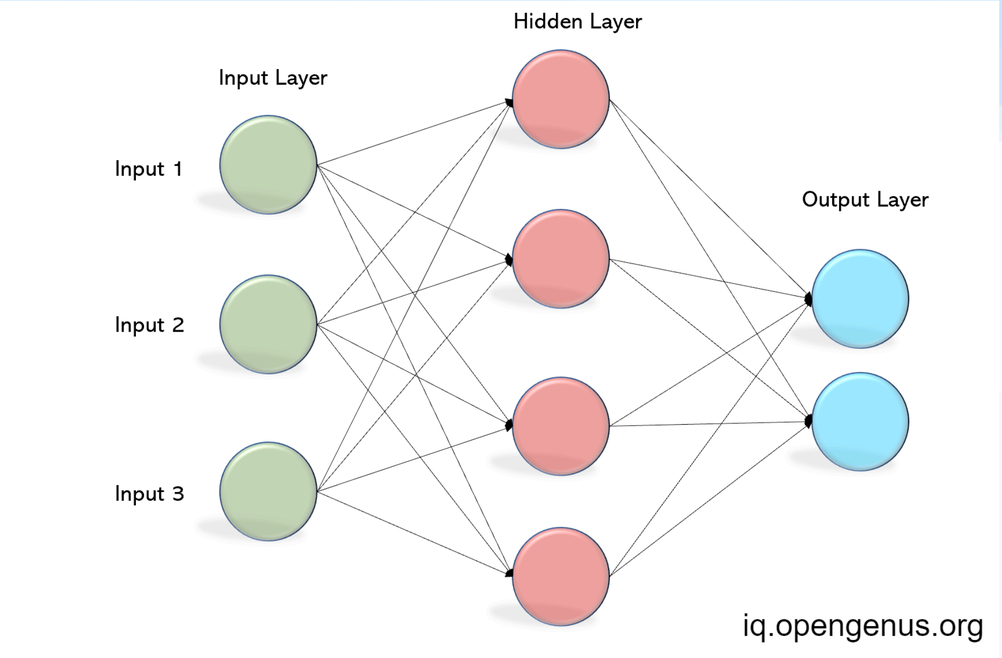



# Aspekte

Aspekte im Output
z.B.
output 1 = Hund/**Katze**
output 2 = stehend/**rennend**

**0,0 = stehender Hund**
**1,1 = rennende Katze**

man sieht:
mit binären Aspekten lassen sich schnell viel **Kombinationen** darstellen.



# Lernen

Die Begriffe **lernen** und **trainieren** bedeuten im Zusammenhang von Deep Learning das **Anpassen der Modell Parameter** ("**Gewichte**") so dass eine Zielfunktion optimiert wird zum Beispiel erhöhen der Genauigkeit oder **Reduzierung des Fehlers.**

# Fehler

Um zu messen wie gut unser Netzwerk die Trainingsdaten modelliert, messen wir entweder die Genauigkeit (**accuracy**), oder den so genannten **Fehler.**

Es gibt verschiedene Arten den **Fehler (error, loss,** cost, risk) eines Netzes zu berechnen.

Neben den **standard** Methoden (absolute **Differenz** abs(**y-ȳ**), $\sum(y_i-\bar{y}_i)^2/N$ mean square error MSE) **Quadrierter Fehler** $\sum(y_i-\bar{y}_i)^2/N$ ist besser, weil große Abweichungen mehr ***bestraft*** werden.

Gibt es auch etwas fortgeschrittene Berechnungen:

- Cross Entropy (next slide)
- Custom Errors (next slide)

💡 Während die **Genauigkeit** stets in einem Bereich **zwischen 0 und 100%** liegt, ist der **Fehler eher abstrakt** und kann beliebige Werte annehmen allerdings mit der Eigenschaft dass er monoton fallen sollte. Geringere Fehler entsprechen idealerweise höherer Genauigkeit.

Generell kann man das Vorzeichen beliebig vertauschen also wenn man eine Messgröße hat welche man nach oben **optimieren** möchte ist es das selbe als wenn man die negative Messgröße nach unten optimiert.
⚠️ Es ist Standard fallende Größen zu verwenden, also Verlust/**loss** zu **reduzieren.**

# Fehler

Um zu **lernen** schauen wir, ob unser output $\bar{y}$ **ähnlich** mit dem label y ist.

Wenn ja: alles gut, Netz kann so bleiben.
Wenn nicht: ändere **Gewichte**.

Die Diskrepanz von Ziel und Aktivierungs-Ausgabe nennt sich **Fehler.**

# Fehler

Um zu **lernen** schauen wir, ob unser output $\bar{y}$ **ähnlich** mit dem label y ist.

Wenn ja: alles gut, Netz kann so bleiben.
Wenn nicht: ändere **Gewichte**.

Generelle Idee: schaue in welche Richtung der Fehler geht und verschiebe Gewichte in andere Richtung (Vorzeichen).

# Fehler

Anpassen der Gewichte:

1) Zufällig! 'Evolution'
wenn das Netzwerk sehr schwer zu berechnen ist,
wenn es nicht 'differenzierbar' ist

1) Generelle Idee: schaue in welche Richtung der Fehler geht und verschiebe Gewichte in andere Richtung (Vorzeichen).

•) Verbunden: bei jeden Schritt ein bisschen zufällig

# Wie Lernen?

Wie können wir unser Netzwerk **lernen** lassen / **trainieren?**

**Anpassen der Modell Parameter** ("Gewichte") so dass eine Zielfunktion optimiert wird zum Beispiel erhöhen der Genauigkeit oder **Reduzierung des Fehlers.**

**Naiv / Zufällig:** (blind) an den Gewichten wackeln und schauen ob der Fehler kleiner wird.

**Mathematisch:** Berechnen, welche Änderung der Gewichte zu kleinerem Fehler führt.

**Automatisch:** python Bibliotheken nehmen uns das Rechnen ab!

**Stochastisch:** dem berechneten Wert ein Stück weit vertrauen, aber nur ein wenig an den Gewichten wackeln um nicht 'übers Ziel hinaus zu schiessen' (siehe Überanpassung Tag 4)

# Aufgaben Perceptron

**a) Berechne für den einfachsten Fall ein Perceptron mit zwei Eingängen und einem Ausgang die lineare Aktivierung bei initialen Gewichten w0 = w1 = w2 = 1/2** (w0 bias)
**und input vector x=(x1,x2)=(1,1)** $y = \sum w*x = 1*w0 + x1*w1 + x2*w2$

**b) Das gewünschte Ergebnis der Aktivierung soll 1 sein. Wie könnte man die Gewichte anpassen? Konkrete Werte für w 'raten'.**

**c1) Das gewünschte Ergebnis der Aktivierung für input (1,1) soll 1 sein.**
**c2) Das gewünschte Ergebnis der Aktivierung für input (1,0) soll 1 sein.**
Vorschlag w=(*-1*,2,2) => x=(1,0) y(1,0) = *-1* + 1*2 +0*2 = -1+2 = 1 'forward pass'
**c3) Das gewünschte Ergebnis der Aktivierung für input (0,1) soll 1 sein.**
**c4) Das gewünschte Ergebnis der Aktivierung für input (0,0) soll 0 sein.**

**d) Welche Funktion lernt unser Netz? die "Oder Funktion" x1 oder x2**

**e) Wie könnte man die Gewichte konsistent anpassen so dass *alle* Bedingungen erfüllt sind?**
**w=(0,1,1) erfüllt alle bis auf c1**
**w=(0,0,1) erfüllt alle bis auf c2**
**w=(0,1,0) erfüllt alle bis auf c3**
**brauchen wir Schwellwert Funktion (z.b. > 0.5)? Kommen wir exakt auf 1 oder reicht uns ≈ 1?**
**w=(0,2/3,2/3) (1,1) => 4/3 Fehler = 1/3 (1,0) => 2/3 Fehler = 1/3 (0,0) = 0 Fehler Null**
**damit minimieren wir den Gesamten Fehler / Globalen Fehler.**
**Ist das der Optimale Fehler, optimale Gewichte unter absolutem Fehler?**

**Nachdem obiges per Hand berechnet wurde, vergleiche mit torch:**

```
model = torch.nn.Linear( 2 , 1, bias=True) # most trivial
```

**Wie könnte man Fehler und Optimierer selber schreiben?**

**Warum kann XOR *nicht* mit einem einfachen Linear Layer gelernt werden?**

# Aufgaben Perceptron

**Anpassen der Gewichte über Fehler**
**am Beispiel**
w=(-1/2,1,1)
x=(1,1)
c1 => 1.5 Ziel = 1

Forward pass über das Ziel hinausgeschossen => Gewichte reduzieren!

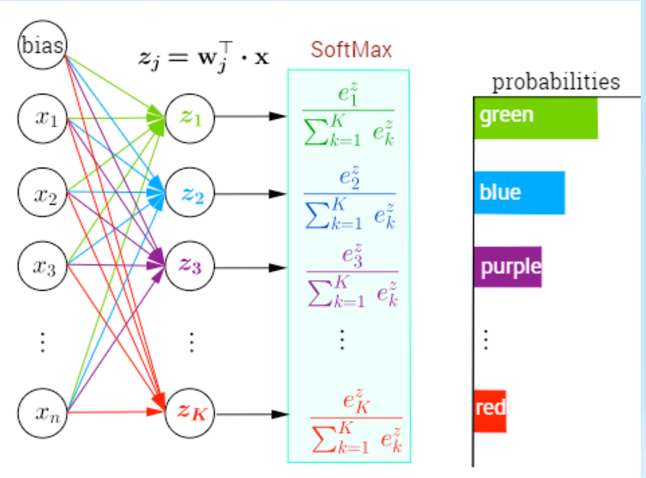
Fehler **-0.5** bei **error**= c1_target=1 - c1_evaluated=**1.5** = -1/2
**Schrittweite** / lr = "Lernrate **0.1**"

Korrigiere w => $\mathbf{w_i} = w_i + \mathbf{lr} * \mathbf{error} * \mathbf{input}_i$ ($input_i = 1$ für i=0 'bias')
w=(-1/2,1,1) = (-1/2,1,1) + (0.1*-.5*1, 0.1*-.5*1, 0.1*-.5*1) = [-0.55, 0.95, 0.95]
Aktivierung mit neuen Gewichten y= 1* -0.55 + 1 * 0.95 + 1* 0.95 ≈ **1.35**
w=(-1,1/2,1/2) c1 = 1

In den nächten Tagen: wie finden wir 'richtige' **Schrittweite um unsere Fehler zu korrigieren, aber nicht über zu korrigieren.**

# Fehler

**CrossEntropy** für '**Klassen**' bewertet zB wie unsere Zahlen 0…9 in verschiedenen **Töpfen** landen.
Hierzu werden am Ende alle Töpfe durch die Summe der Aktivierungen geteilt, was man **Softmax** nennt. Das erinnert an **Wahrscheinlichkeitsvariablen**, deren **Summe stets 1** ergeben.




**10 Zielvariablen haben Zustände 0/1, praktisch dazwischen [0,1] (ja ist Ziffer '7', oder nicht, oder vielleicht)**

Eine weiche (arg)maximum Funktion schaut, in welchem der Töpfe der größte Wert ist.
Ein gutes Maß zum Schauen, ob die Werte in unseren Töpfen $p_i$ unseren gewünschten Werten $y_i$ entsprechen ist

**Normal: $y_i * \bar{y}_i$ solange bei einem Zustand beide 1 sind, kommt 1 raus, was gewünscht/anzustreben ist.**
**wir formulieren es als Minimierungsproblem um:**
CrossEntropy (logistische Verlustfunktion) $-\sum y_i \log(p_i)$
(log ist monoton, man verliert nur Skalierung, lässt sich leichter ableiten!)
Der Cross-Entropy-Verlust ist effektiv, weil er große Strafen für sicher vorhergesagte falsche Antworten verhängt, was die Konvergenzgeschwindigkeit des Lernprozesses beschleunigt.

# Custom Errors

Neben der Nähe zum gewünschten Ergebnis kann man auch andere **(meta)Kriterien** in den "**Fehler**" mit einbauen:

Man kann fast alles **messen** und **bewerten => bestrafen** oder **belohnen.**

**besserer_Fehler = Fehler + a * custom_Fehler  (z.B. 7≠2 1≠7)**

In der Tat kann man mit einer eigenen Fehler Funktion das Verhalten vom Neuronalen Netz fein steuern ("**Belohne** glatte Kurven" "**Bestrafe** krakelige Schrift" )

Dazu genügt es oft verschiedene **Maße** zu **addieren,** mit **Gewichtungsfaktor** (nachdem sie mit einem Faktor auf ähnliche **Größenordnungen** gebracht wurden)

Der Optimierer reduziert den Fehler dann langfistig so, dass alle Kriterien ähnlich gut erfüllt werden.

# Back Propagation

Wenn wir einen Fehler berechnet haben können wir sehen welche **Gewichte** den **größten Einfluss** auf den **Fehler** hatten und die Gewichte dementsprechend anpassen. Präzise funktioniert das über die sogenannte **Backpropagation**, also **Rückwärtsausbreitung** oder Rückverfolgung des Fehlers.

Mathematisch sagt uns die **Ableitung** wie stark die Sensitivität des Fehlers bezüglich kleiner **Änderungen** dieser **Gewichte** ist. Um die Ableitung zu berechnen erinnern wir uns an die einfache

**Kettenregel:** g(f(x))' = g'(f(x)) * f'(x)

Also egal wie tief wir im Netz sind, welches für einen input x einen output y berechnet, wir müssen nur mehrfach y=g(f(x)) betrachten.

Wie stark die Gewichte tatsächlich angepasst werden hängt vom Optimierer ab, welcher zum Beispiel die **learning rate** mit der Änderung multipliziert.

# Back Propagation

**Backpropagation** (Rückwärtsausbreitung) ist ein zentrales Konzept in der Funktionsweise tiefer neuronaler Netze und bezieht sich auf den Prozess, durch den Gewichte in einem Netzwerk angepasst werden, um den Fehler zu minimieren.
Es nutzt den **Gradientenabstieg** (Gradient Descent),
um den **Fehler Schritt für Schritt zu reduzieren,**
indem es die **Ableitungen der Verlustfunktion** (Loss Function) bezüglich der Gewichte berechnet.

Backpropagation ist ein Verfahren, das den Fehler (Loss) in einem neuronalen Netz rückwärts durch die Schichten propagiert, um die Gewichte anzupassen. Dies geschieht in zwei Schritten: Forward Propagation und Backward Propagation.

- **Vorwärtsausbreitung** (**Forward** Propagation):
  Die Eingabedaten werden durch das Netzwerk geleitet, um eine Vorhersage zu generieren.
  Der **Fehler (Loss)** zwischen der Vorhersage und dem tatsächlichen Zielwert wird berechnet.
- **Rückwärtsausbreitung** (**Backward** Propagation):
  Der Fehler wird rückwärts durch das Netzwerk geleitet,
indem die Gradienten der Verlustfunktion in Bezug auf die Gewichte berechnet werden.
  Diese Gradienten werden verwendet, um die **Gewichte zu aktualisieren,**
sodass der Fehler bei der nächsten Vorhersage reduziert wird.

# Back Propagation

Probleme der Kettenregel (gelöst!):

Vanishing / Exploding Gradient

https://karpathy.medium.com/yes-you-should-understand-backprop-e2f06eab496b

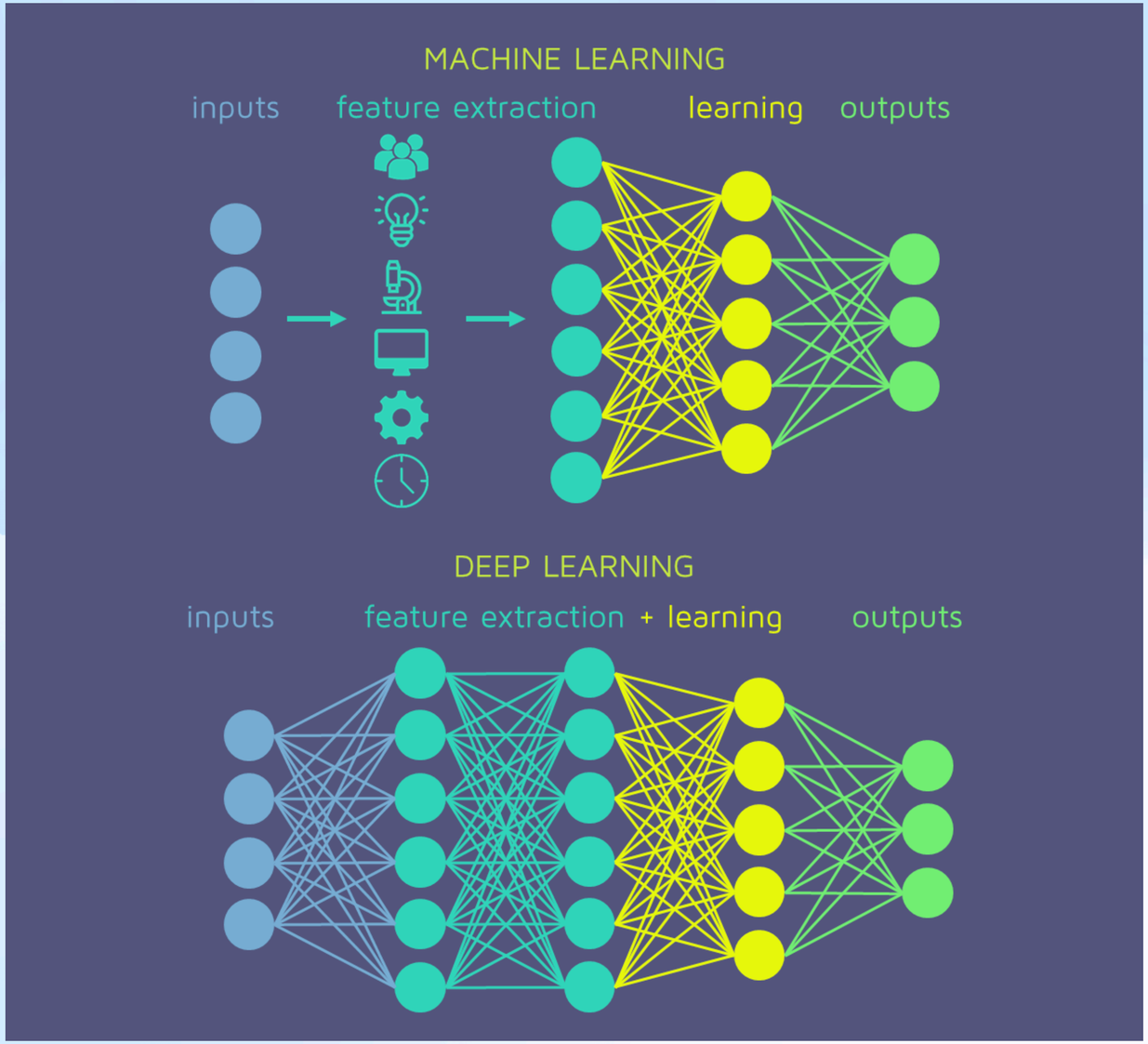
Mathematisch sagt uns die Ableitung wie stark die Sensitivität des Fehlers bezüglich kleiner Änderungen dieser Gewichte ist. Um die Ableitung zu berechnen erinnern wir uns an die einfache **Kettenregel:**
f(g(x))' = (f.g)' = g' * f'g

$y = f( \sum w_i * x_i + w0)$
$g(x) := \sum w_i * x_i + w0$
$y = f( g(x) )$

f unsere Aktivierungsfunktion max(0,x)

f'(x) = 0 für x<0  1 für x>=0
$g'(x_i) = w_i$

# Deep Learning



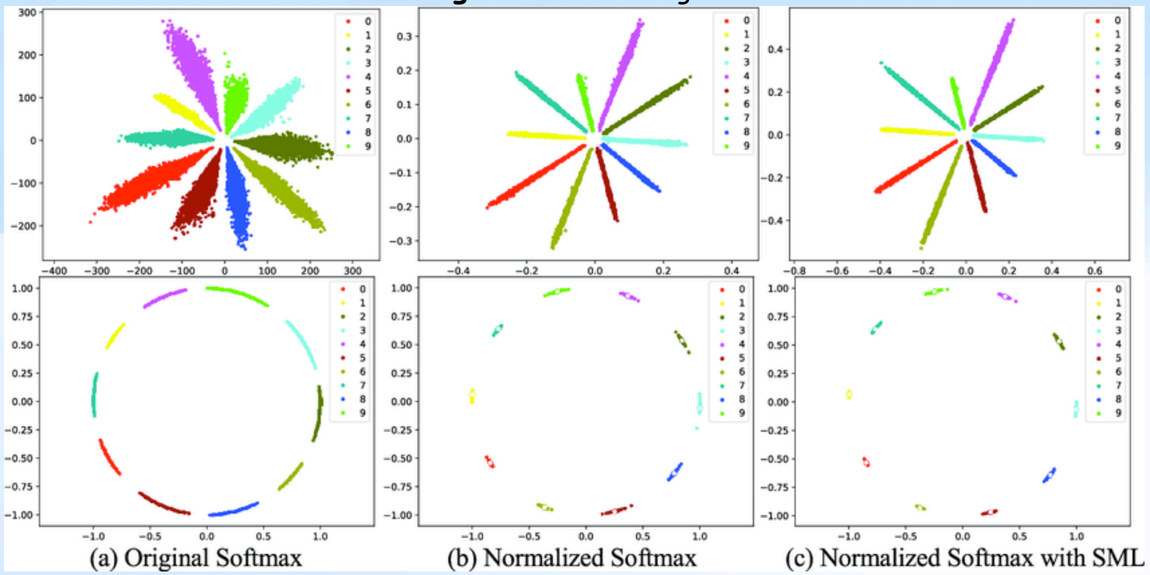
# (Un/Semi-)Supervised

Wenn wir zu Daten die passenden Beschreibungen ("**Labels**") haben, spricht man von **Supervised learning.**

Oft hat man aber nur Roh Daten ohne Label.

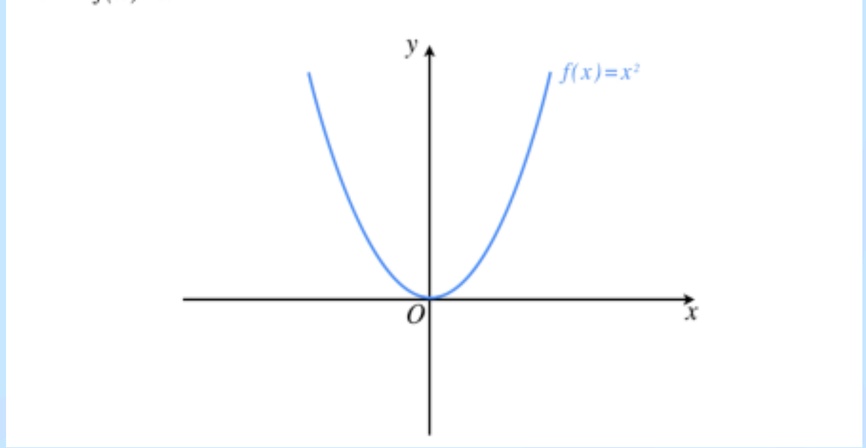
Auch mit diesen lässt sich lernen, dann spricht man von **unsupervised learning:**

- **Kompression n-input n/x-schicht**
- **Generierung:** Das Modell kann lernen die Daten zu reproduzieren
- automatische **Clusterung** Entdeckung von **Mustern und Strukturen** in den Daten.




Mischvarianten nennt man **Semi**Supervised, z.B. wenn **GPT** unvoreingenommen das ganze Internet durchliest (unsupervised) aber dann im letzten Durchgang noch Label bekommt, was 'wahrer' 'hochwertiger' 'sozial akzeptabler' … Content ist.

# Verfahren ohne Ableitung




**Pseudo Ableitung**
**Fehler Funktion = (y-y')^2**
**f(x)=x^2**
**wie findet man <u>blind</u> das <u>Minimum</u>**

**gehe einen Schritt nach links oder rechts**
**rechts: Wert wir grosser => drehe um gehe doppelt so weit zurueck**
**links: Wert wird kleiner => gehe weiter nach links**
**random descent**

**Etwas geplanter (und mathematischer) Minimum finden: Gradient descent**
**f'(x) wenn die Steigung groß ist gehe vorsichtig kleine Schritte**
**f'(x) wenn die Steigung klein ist gehe *mutig* große Schritte**

y=f(x)
Fehler = |y'-y|
**Forward-Forward-Network** (von **Hinton**)

Nachteil: Funktioniert erstaunlicherweise,
aber ist viel langsamer als Gradienten Backpropagation.

Bei Analogen oder **Photonen** Chips wieder relevant.
Problem: Elektronen und Photonen in einem Chip

# AI Grundverständnis

**Legendär verständlich sind die online Kurse von Andrew Karpathy (youtube/blogs)**

**Auf verschiedenem Niveau: Von Anfänger bis Experte (trotzdem recht verständlich)**

**Koryphäen:**

- **Geoff Hinton**
- **Andrew Ng**
- Yann LeCun
- Yoshua Bengio
- Jürgen Schmidhuber (hat alles schon 1990 erfunden)

**(es ist es Wert, von jedem das beste Paper / Video anzusehen)**

# Breiter Output

Wir schreiben die Summen nur in Notation gemeinsam:

$$y_i = \sum w_{ij} x_j$$

**Ein Beispiel:**

input = [1,2]

$weights_1$ = [1,2,3] Gewichte für output $y_1$

$weights_2$ = [2,3,4] Gewichte für output $y_2$

$$y_i = w_i \circ x$$

# Breiter Output

**Die Aktivierungsfunktion wird auf jeden Eintrag im output Vektor einzeln angewendet:**

**Einfaches y ohne Aktivierungsfunktion**

$$y = M*x$$

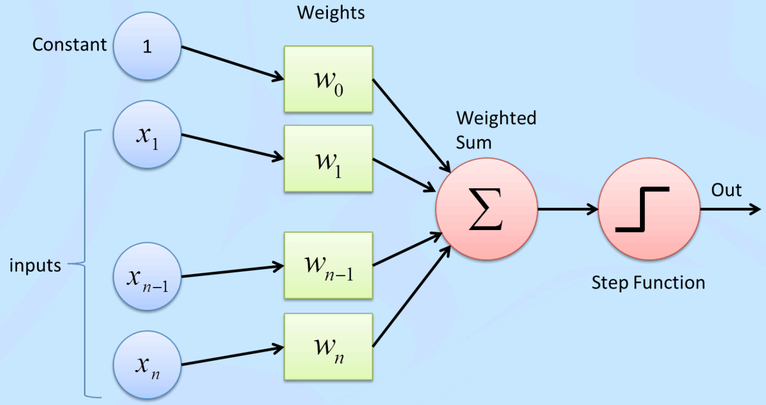
**Besseres y mit Aktivierung nachgeschaltet "$y^+$"**

$$y^+ = g(M*x) = g(y_i)$$

**"Elementweise auf jeden Eintrag im Vektor"**

# Breiter Input

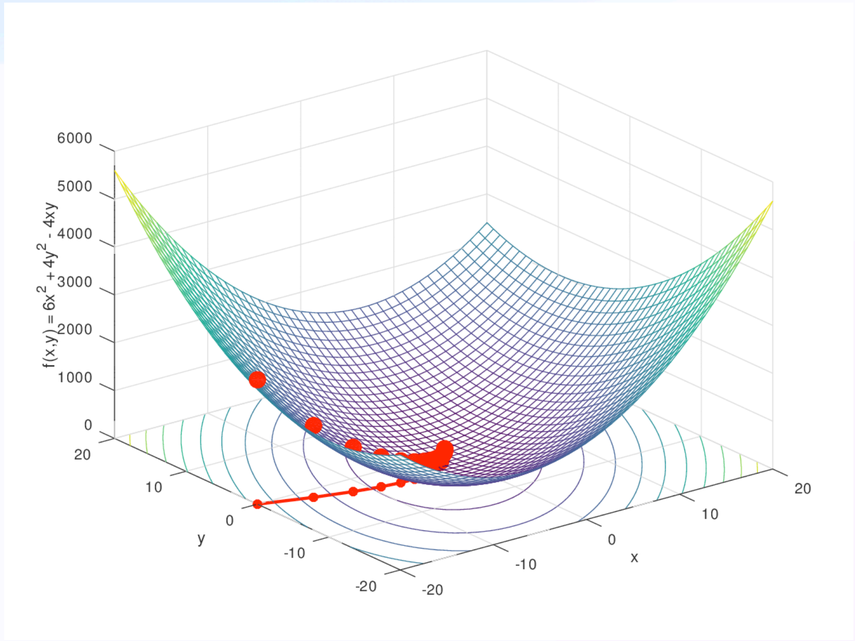
**Wie wir gesehen haben bekommt jedes 'Neuron' den Input von mehreren 'Synapsen':**




**Damit ist unsere Aktivierung Funktion also mathematisch eine Funktion mehrere Variablen.**
**f(x1,x2,…,xn)=…**

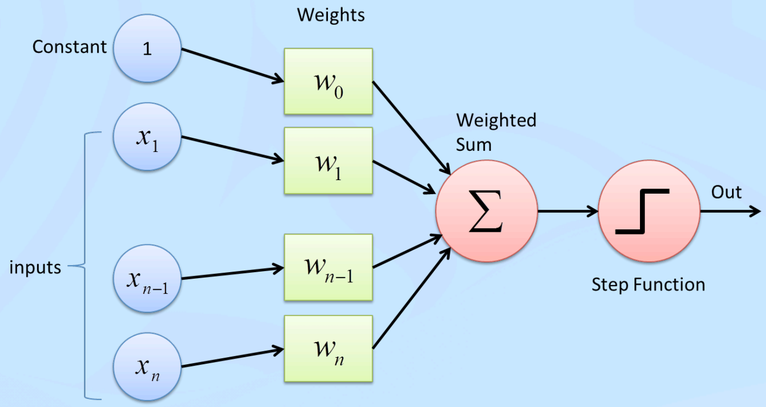
**z.B. f(x1,x2)=x1*x2 oder f(x1,x2) = 2*x1 * x2 * x2**

**Falls das jemand nicht in der Schule**
**oder danach gesehen hat, solche Funktionen**
**kann man sich vorstellen wie Gebirge über der Ebene:**



# Breiter Input : Batch

**Batch: berechne die Aktivierung von mehreren Inputs gleichzeitig!**




**Stapel von Input Vektoren wird parallel ans Netz gegeben.**

**Damit y = W*X mit Inputs als Matrix gesammelt (statt y = W*x input vector x)**

# Gradient

**Gradient = Ableitung von Funktion mehrere Variablen**

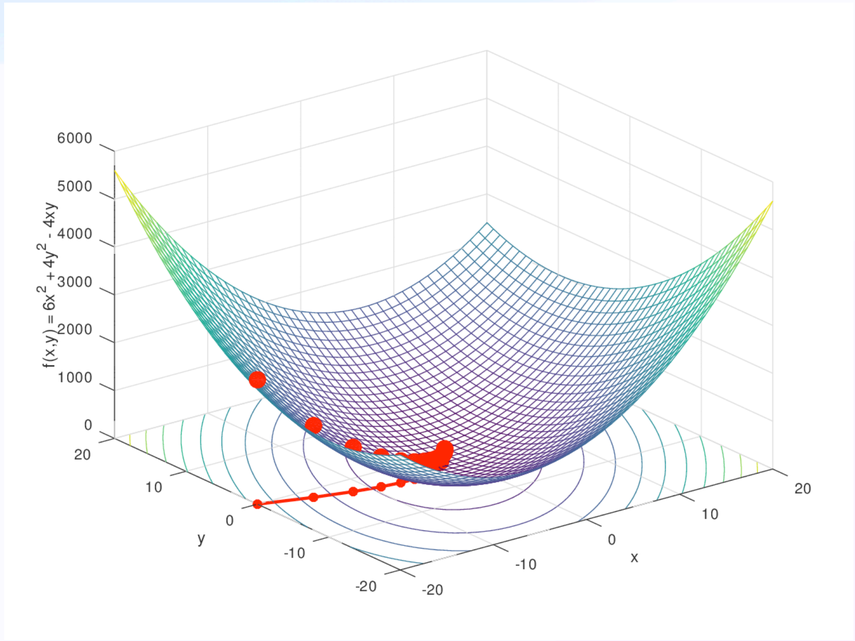
**Wenn eine Funktion von mehreren variablen abhängt kann man sie in verschiedene 'Richtungen' ableiten, also für jede Variable die Ableitung bilden. Diese Ableitungen zusammengefasst als Vektor nennt sich Gradient. Man schreibt üblicherweise ∇f statt f':**

**z.B.**
$f(x_1,x_2) = x_1 * x_2$
$\nabla f = [df/dx_1, df/dx_2] = [x_2, x_1]$

$f(x_1,x_2) = x_1 + 2 * x_2$
$\nabla f = [df/dx_1, df/dx_2] = [1 + 0, 0 + 2] = [1, 2]$

$f(x_1,x_2) = 2*x_1 * (x_2 * x_2)$
$\nabla f = [df/dx_1, df/dx_2] = [\ 2 * x_2 * x_2,\ 2*x_1(2*x_2)]$

💡 **Sowie die Ableitung als tangential Gerade bei einer einfachen Funktion den Grad des Anstiegs oder Abstieg anzeigt, zeigt der Gradient im Gebirge in die Richtung des stärksten Abstiegs beziehungsweise Anstiegs.**



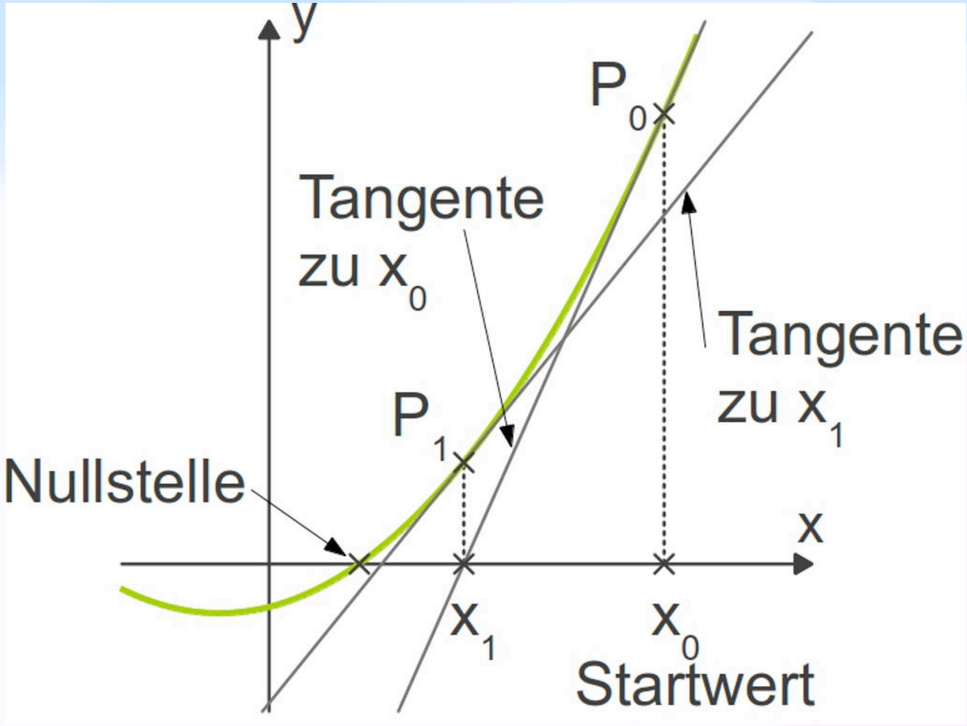
# Modell

Parameter (freie Variablen)
Feste Werte (bias)
Optimierung
Training: Anpassen an Daten

Einfachstes Modell: ein bit "ja/nein"

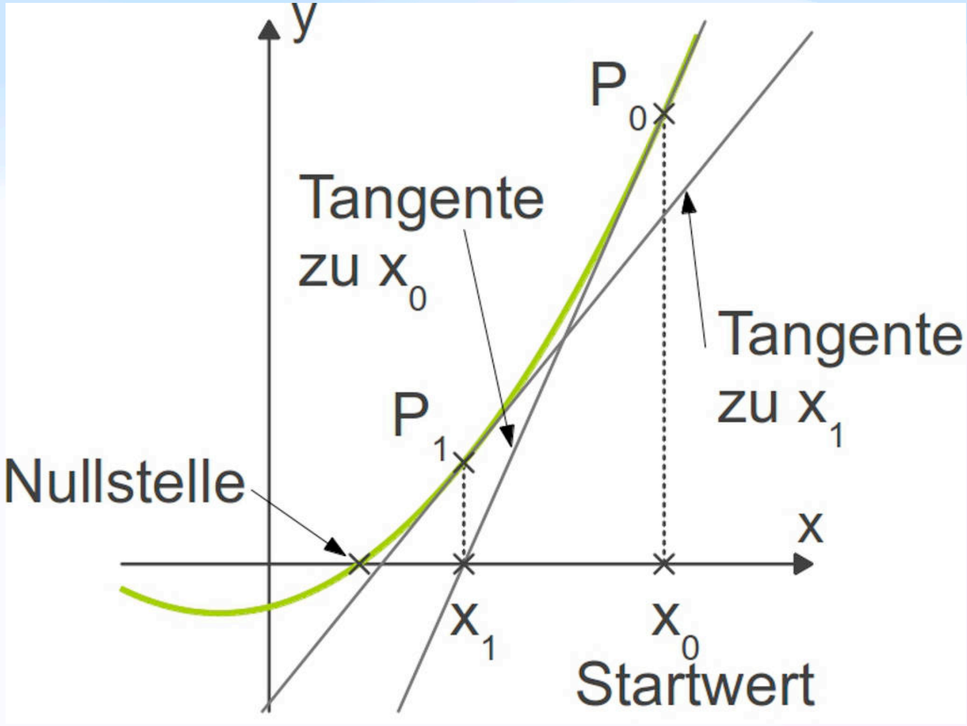
# Newton Verfahren

```
Idee: gehe in Richtung der größten Änderung (Ableitung)

# vereinfachtes newton verfahren
while fehler(x) > 0.01:
    ableitung_von_fehler = 2*x # Tangente
    x = x - ableitung_von_fehler * Schrittweite
    print(fehler(x))
```



# Newton Verfahren in R^n

```
Idee: gehe in Richtung der größten Änderung (Ableitung)

# vereinfachtes newton verfahren
while fehler(x) > 0.01:
    gradient_vom_fehler = …
    x = x - gradient_vom_fehler * Schrittweite
    print(fehler(x))
```



# Aufgaben

**Berechnung von mehreren Aktivierungen gleichzeitig mit EINER OPERATION (mit Python):**

```
y=tanH(W*x)
 input = [1,1]
e)    weights = [0,0,0]
f)    weights = [½,½,½]
g)    weights = [½,½,½]
h)    weights = [1,2,3]

dann für
y=sigmoid(W*x)
```

**In python berechnen, so wie vorhin nur mit Aktivierung.**

Gradient $\nabla f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$

# Aufgaben

**Vorblick: Tensor Operationen**

**Berechnung von mehreren Aktivierungen gleichzeitig mit EINER OPERATION (mit Python):**

**y=tanH(W*X)**
**e) input = [1,1] weights = [0,0,0]**
**f) input = [0,0] weights = [½,½,½]**
**g) input = [½,½] weights = [½,½,½]**
**h) input = [1,2] weights = [1,2,3]**

**X = [[1,1,1],[1,0,0],[1,1/2,1/2],[1,1,2]]**
**W = [[0,0,0],[1/2,1/2,1/2],[1/2,1/2,1/2],[1,2,3]]**

**so wie e...f nur mit y=sigmoid(∑$w_i$*$x_i$)**

Gradient $\nabla f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$

# Aufgaben

**Berechnung von Aktivierungen**

**1) Manuell**
**Modell Linear : $y = \sum w_i * x_i + w_0$  f(input, weights)=…**
**a) input = [½,½] $w_0$=½ weights = [½,½]**
**b) input = [1,2] weights = [1,2,3] (erstes Element w0 ist bias)**

**Modell Linear mit Aktivierung : $y=max(0,\sum w_i * x_i)$ (x0 ist 1 per convention, für bias w0)**
**c) input = [-½,½] weights = [½,-½,-½]**
**d) input = [1,2] weights = [-1,-2,-3]**

**Modell Linear mit Aktivierung : $y=tanH(\sum w_i * x_i)$**
**e) input = [1,1] weights = [0,0,0]**
**f) input = [0,0] weights = [½,½,½]**
**g) input = [½,½] weights = [½,½,½]**
**h) input = [1,2] weights = [1,2,3]**

**2) Mit Python**
**so wie e...f nur mit $y=sigmoid(\sum w_i * x_i)$**

Gradient $\nabla f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$

# Aufgaben Gradient

Hier sind einfachere Aufgaben zur Wiederholung des Gradienten:

1. **Gradient einer einfachen Funktion:**

   Bestimmen Sie den Gradient der Funktion $f(x, y) = x^2 + y^2$.

2. **Gradient in einem Punkt:**

   Berechnen Sie den Gradient der Funktion $f(x, y) = 3x + 4y$ am Punkt $(2, -1)$.

3. **Gradient einer linearen Funktion:**

   Finden Sie den Gradient der Funktion $f(x, y) = 5x - 2y$.

4. **Gradient einer quadratischen Funktion:**

   Bestimmen Sie den Gradient der Funktion $f(x, y) = x^2 - 2xy + y^2$.

```
1. Δf(x,y) = [df/dx, df/dy] = [2x, 2y]

# Für uns praktische Gradienten (Fehler) :
# Name der Variablen schon mal so gewählt, dass y=output t=target
f(y,t) = 1/2(y-t)² # MSE
Δf

f(y,t) = |y-t|
Δf

f(y,t) = f(y₁,y₂,t₁,t₂) = 1/n(y-t)² = 1/n∑(yᵢ-tᵢ)  # MSE als vector
Δf
```

# Aufgaben Gradient

Hier sind einfachere Aufgaben zur Wiederholung des Gradienten:

1. **Gradient einer einfachen Funktion:**

   Bestimmen Sie den Gradient der Funktion $f(x, y) = x^2 + y^2$.

2. **Gradient in einem Punkt:**

   Berechnen Sie den Gradient der Funktion $f(x, y) = 3x + 4y$ am Punkt $(2, -1)$.

3. **Gradient einer linearen Funktion:**

   Finden Sie den Gradient der Funktion $f(x, y) = 5x - 2y$.

4. **Gradient einer quadratischen Funktion:**

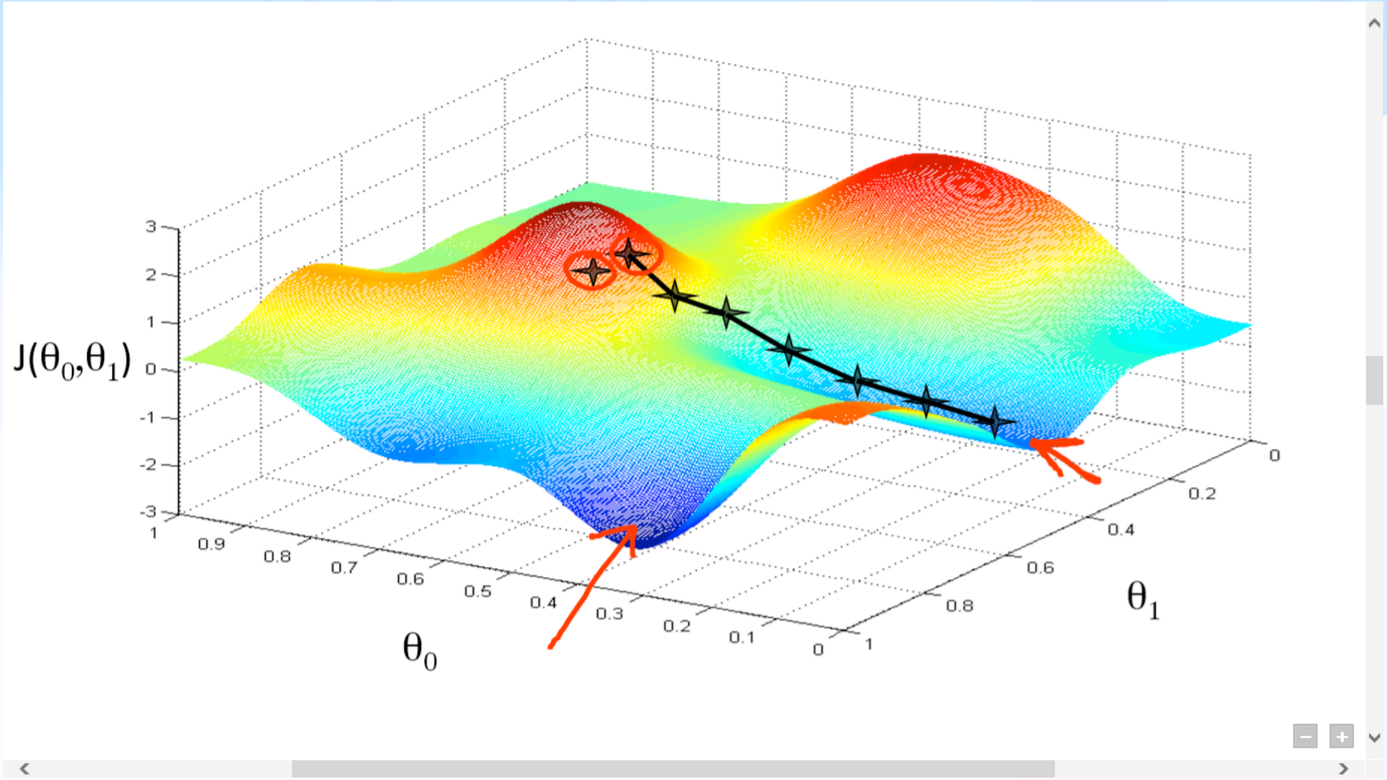
   Bestimmen Sie den Gradient der Funktion $f(x, y) = x^2 - 2xy + y^2$.

```
1. Δf(x,y) = [df/dx, df/dy] = [2x, 2y]

# Für uns praktische Gradienten (Fehler) :
# Name der Variablen schon mal so gewählt, dass y=output t=target
5. f(y,t) = 1/2(y-t)² = 1/2 (y^2 - 2*t*y + t^2) # MSE
Δf(y,t) = [df/dy, df/dt] = [1/2*2y - 1/2*2t , - 1/2*2*y + 1/2*2*t ] = [ y - t, t - y ]

6. f(y,t) = |y-t|
Δf = sign(y-t) == 1 if y-t >0 , -1 if y-t<0

f(y,t) = f(y₁,y₂,t₁,t₂) = 1/n(y-t)² = 1/n∑(yᵢ-tᵢ)  # MSE als vector
Δf
```

# Gradient

So wie wir im Newton Verfahren als Beispiel bei der f(x)=x^2 Funktion zum Minimum gefunden haben, können wir mit dem Gradienten bei allgemeineren Funktionen die tiefste Stelle im Gebirge finden. Übertragen auf unsere Fehlerfunktion können wir also somit unseren Fehler minimieren.

Das ist die Grundidee von Backpropagation, im Detail erörtern wir das am Donnerstag unter dem Begriff Stochastik Gradient Descent (SGD).



# Buch

```
Seite 122 Optimierung
Seite 163 ff 204 morgen lesen

Anhang B - Autodiff
```

**Differenzierung von Hand**

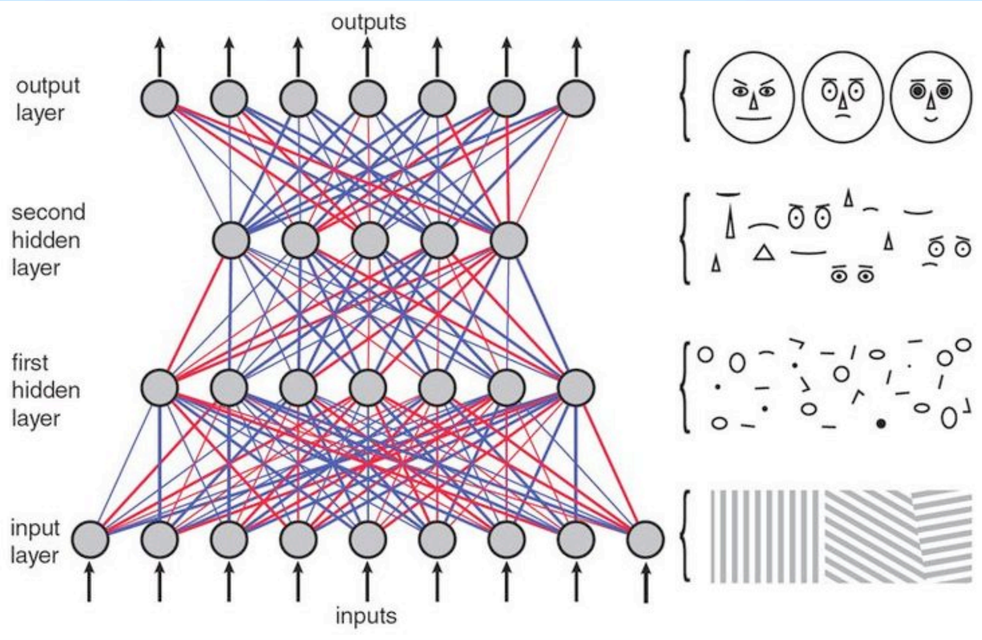
| Unterstützung | IDE | Framework |
|---|---|---|
| VOLL | PyCharm | PyTorch |
| Partiell | VSCode | Tensorflow |
| Kaum | andere | andere |

# Tiefe / Breite

**Layer = Schicht = einmal Matrix plus Aktivierung**

**Breite = Anzahl der Aktivierungen pro Schicht**

**Tiefe = wie viele Schichten (hier 2-4)  meist Anzahl der Matrizen Multiplikationen, also hier 3**




**Kann Breite von hidden Schicht kleiner sein als input / output?**

**Erlaubt / gut? ja / ja, "Bottleneck" bei *Kompression, Feature extraction, gegen overfitting, für Abstraktion on Generalisierung***

**Kann Breite von hidden Schicht großer sein als input / output?**

**Erlaubt / gut? ja / ja bei Erhöhung der memory capacity, bei Transformer associations, Zusammenhänge erkennen, Features in verschiedenen Konstellationen  "Mehr hilft mehr"**

# Tiefe / Breite

**Layer = Schicht = einmal Matrix plus Aktivierung**

**Breite = Anzahl der Aktivierungen pro Schicht**

**Tiefe = wie viele Schichten (hier 2-4)**

**Kann Breite von hidden Schicht kleiner sein als input / output?**

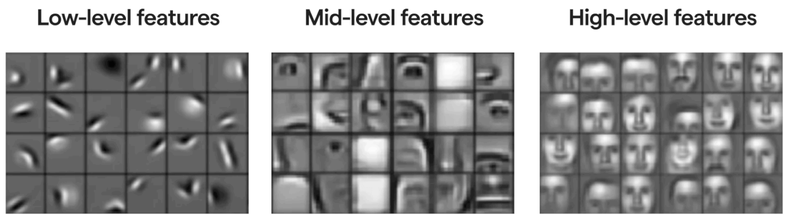
**Erlaubt / gut?** ja / ja, "Bottleneck" bei *Kompression, Feature extraction, gegen overfitting, für Abstraktion on Generalisierung*

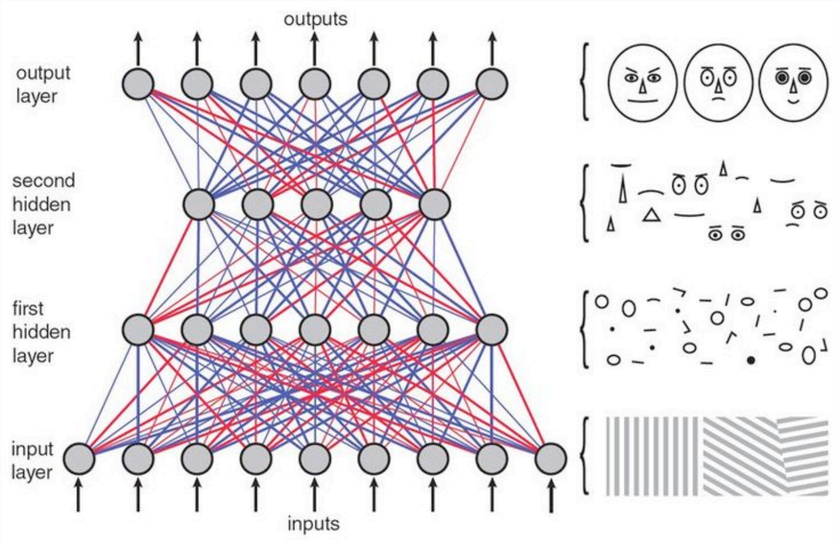
Kann Breite von hidden Schicht großer sein als input / output?

**Erlaubt / gut?** ja / ja bei Erhöhung der memory capacity, bei Transformer associations, Zusammenhänge erkennen, Features in verschiedenen KonstellationenKönnen wir beliebig tiefe Netze erstellen? Erlaubt / gut? Kosten / Nutzen

**Generell: "Mehr hilft mehr" geht seit ≈ 15 Jahren**

**Mittlere Schicht: Abstrakt**



# Instabilitäten

**Bis vor ≈ 15 Jahren hatte man noch große Probleme mit Instabilitäten:**

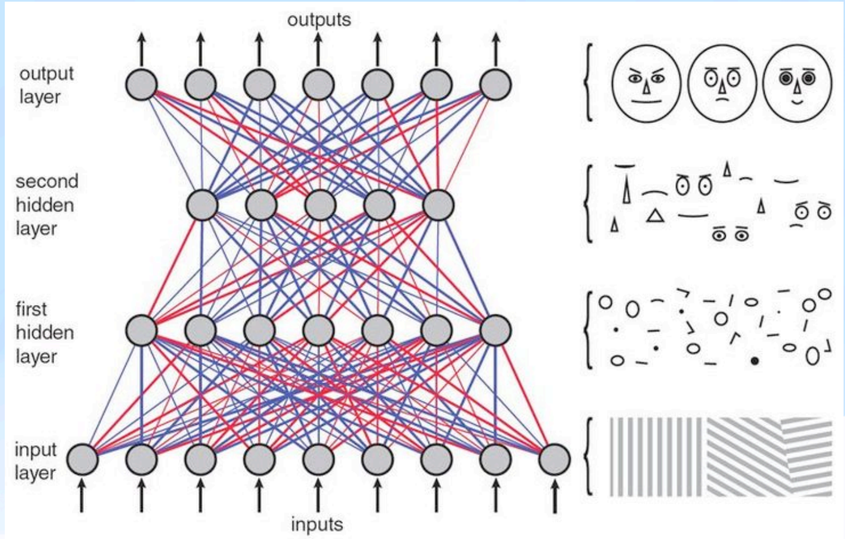
**<u>Vanishing</u> Gradient: die Fehlerinformation geht in zu tiefen Schichten gegen 0**

**<u>Exploding</u> Gradient: die Fehlerinformation schaukelt sich auf und geht gegen ∞**

**Mitigation / Gegenmaßnahmen:**

**Neue "Regulierungs-"Erfindungen:**

**residual/skip connections, dropout, layer norms, …**



# Breiter Output

Idee: reagiere auf verschiedene Eingaben (input)
mit mehreren verschiedenen outputs gleichzeitig!

Dann hat man gleichzeitig nicht nur eine Summe, sondern mehrere.

💡 **Jede Summe für sich ist genau so einfach wie vorher**

Wir schreiben die Summen nur in Notation gemeinsam:

$$y_i = \sum w_{ij} x_j$$

**Das hat den Vorteil, dass man die Summen als Matrizen Multiplikation zusammenfassen kann. Rechnerisch ist es aber identisch dazu ein paar Summen per Hand zu rechnen!**

$$y = \underline{W} * x$$

# Vokabular

**Perceptron** **(Synonyme: Fully Connected FC, Dense Layer, Linear Layer)**
**Layer = Schicht**
**Netz Parameter = Gewichte**
**Bias (Verschiebung des Schwellwertes)** auch ein Gewicht (input immer '1')
**Hyperparameter** (Learning Rate, Schrittweite) z.B. (Anzahl der Schichten)
**Aktivierungsfunktionen (relu, tanh, sigmoid, …)**
**Forward step: Berechnung der Aktivierung / output**
**Backward step: Anpassung der Gewichte**
**$\bar{y}$ = Aktivierung (vom Perceptron / vom Netz ) ≈ output ≈ prediction**
y = **Label, target** Zielparameter

elementweise (z.b. Aktivierungsfunktion)
**Fehler (error, loss,** cost, risk)
**Optimierung**
**Lernen/Training**
**Modell = Neuronales Netz**

Backpropagation ≈ Gradientenabstieg am Donnerstag, Fehlerminimierung

Attribut ≈ Feature ≈ Aspekt ≈ Eintrag im input/output Vektor?
Architektur: bisher nur **Anzahl** von Schichten, **Breite / Size** von hidden layers