Warum eine Gerade nicht reicht (Aktivierungsfunktionen)

Kapitel online

Zehn Schichten ohne Aktivierung sind immer noch eine Gerade. Erlebe live, wie erst der ReLU-Knick ein Netz mächtig macht — und baue aus drei Knicken selbst eine Kurve.

Praxis · Kapitel 19 von 50 · ca. 15 Min. · 200 XP

Einfach gesagt: Gerade auf Gerade bleibt eine Gerade — erst ein einfacher Knick, der alles Negative zu null macht, lässt dein Netz echte Kurven lernen.

In „Polynomiale Regression“ hat dein Netz eine Kurve gelernt. Da liegt die Frage nahe: *„Warum dann nicht einfach IMMER mehr Schichten stapeln?“* Die überraschende Antwort: **Mehr Schichten allein bringen exakt gar nichts.** Was ein Netz wirklich mächtig macht, ist ein winziger Knick — die **Aktivierungsfunktion**.

Was du hier lernst

- Warum **Gerade + Gerade** wieder nur eine Gerade ergibt — Tiefe allein bringt nichts
- Was **ReLU** rechnet: max(0, x) — ein Knick bei null, mehr nicht
- Wieso sich aus verschobenen Knicken **jede Kurve** bauen lässt (Kurven-Lego)
- Dass die Aktivierung **elementweise** auf den Tensor jeder Schicht wirkt
- Wo **Sigmoid und Tanh** sitzen — und warum ReLU die versteckten Schichten gewonnen hat

Für den Unterricht

Einstieg (3 Min): Kapitel gemeinsam lesen, Alltagsbeispiel aufgreifen.

Erkunden am Gerät (9 Min): die interaktiven Module auf herrki.de/145_warum_nichtlinear ausprobieren.

Sichern mit Quiz (3 Min): „Kurzer Check" besprechen, „Das nimmst du mit" festhalten.

Die unbequeme Mathematik: Gerade + Gerade = Gerade

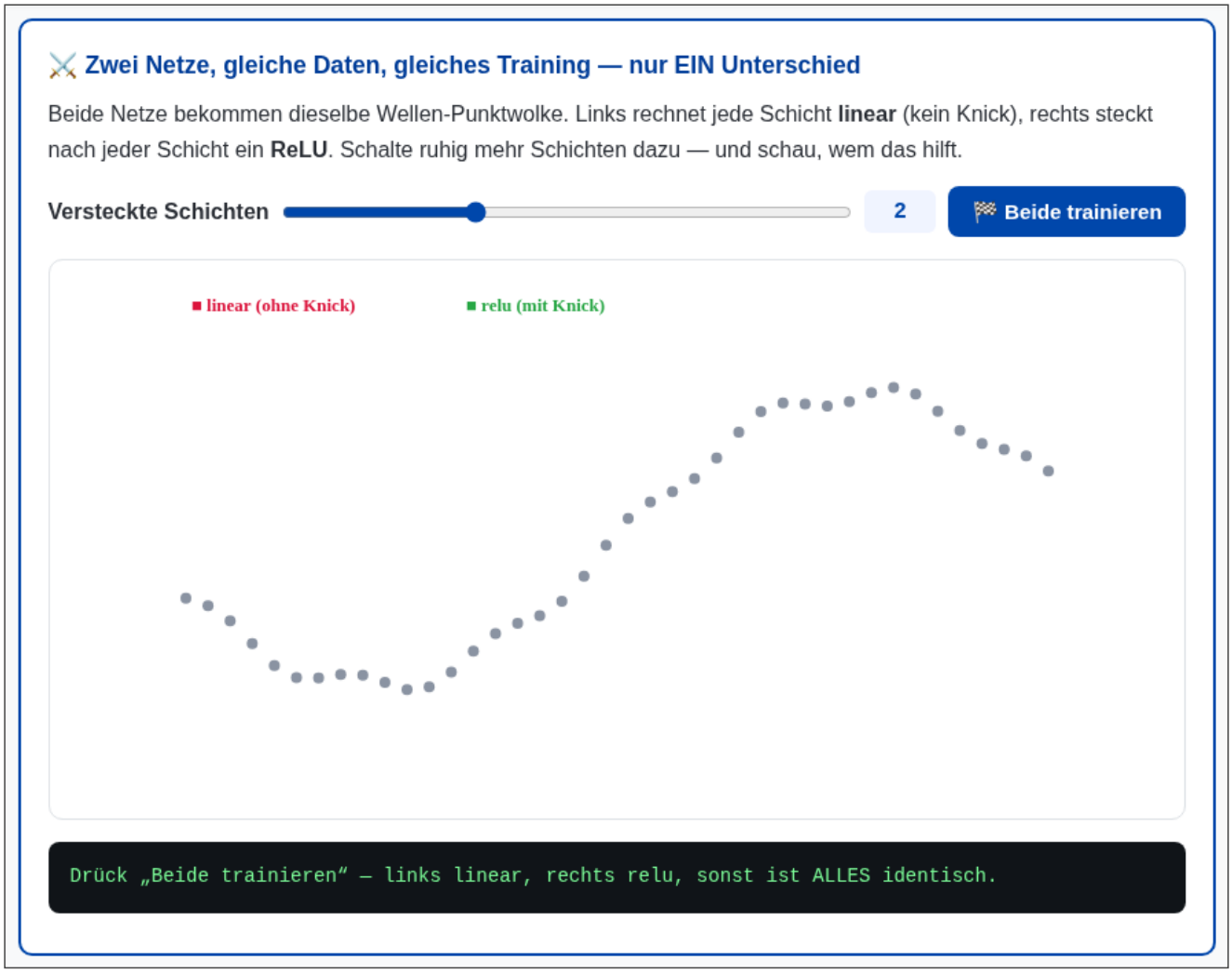
Eine Schicht ohne Aktivierung rechnet *Ausgabe* = *Gewicht* × *Eingabe* + *Bias* — eine Gerade. Stapelst du zwei davon, rechnest du „Gerade von einer Geraden“:

```
// Schicht 1: y = 2x + 1 Schicht 2: z = 3y − 4
z = 3·(2x + 1) − 4 = 6x − 1 // ... wieder nur eine Gerade!
```

Das gilt für 2 Schichten, für 10, für 100: **Ohne Knick kollabiert jeder Stapel zu einer einzigen Geraden.** Glaubst du nicht? Musst du nicht — gleich siehst du es live.

Am Gerät ausprobieren: herrki.de/145_warum_nichtlinear#abschnitt-2

Direkt zum Modul



Interaktives Modul – am Gerät ausprobieren

Was macht dieser ReLU eigentlich?

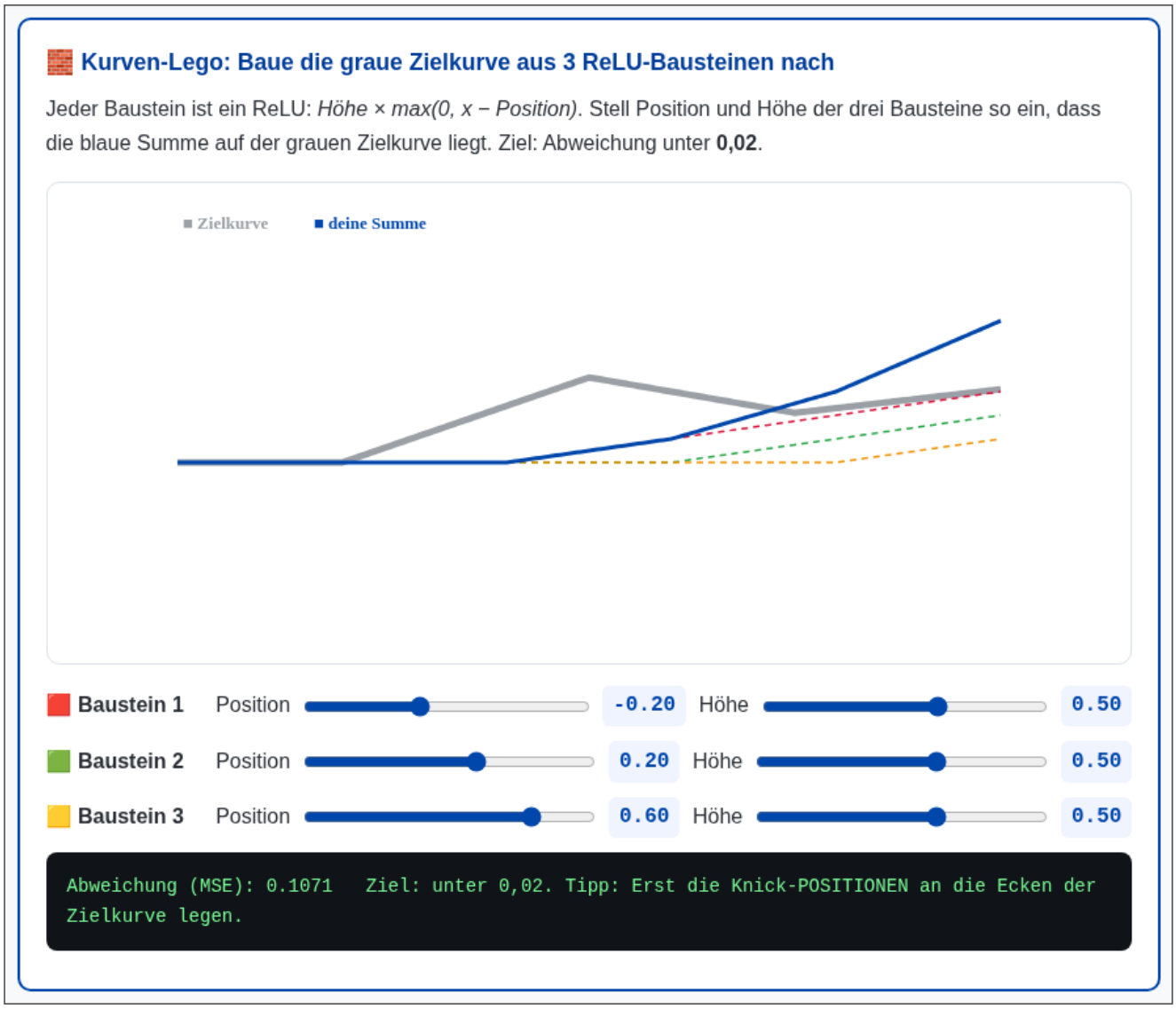
ReLU („Rectified Linear Unit“) ist fast schon beleidigend simpel:

```
relu(x) = max(0, x) // negativ? -> 0. positiv? -> durchlassen. Das ist ALLES.
```

Ein Knick bei null. Aber: **Summen von verschobenen, skalierten Knicken können jede Form annehmen.** Beweis zum Selberbauen:

Am Gerät ausprobieren: herrki.de/145_warum_nichtlinear#abschnitt-3

Direkt zum Modul



Interaktives Modul – am Gerät ausprobieren

Und jetzt der Gedankensprung: Du hast gerade mit 3 ReLU-Bausteinen von Hand gemacht, was eine Schicht mit 128 Neuronen automatisch tut — nur mit 128 Bausteinen, deren Positionen und Höhen per Gradient Descent gelernt werden. Deep Learning ist Kurven-Lego in Millionen-Teile-Packung.

Runter zum Tensor

Die Aktivierung wird **elementweise** auf den Ausgabe-Tensor jeder Schicht angewendet (Kapitel 060): jede Zahl einzeln durch den Knick. Nehmen wir dein Kurven-Lego von eben — drei Bausteine an den Positionen −0,2 / +0,2 / +0,6, jeder mit Höhe 0,5 — und schauen an *einer* Stelle nach, bei **x = 0,4**:

```
Schicht 1 rechnet x − Position, ein Wert je Baustein:
roh [3]: 0,4−(−0,2) 0,4−(+0,2) 0,4−(+0,6)
         = +0,60 +0,20 −0,20

relu() nimmt jede Zahl EINZELN, max(0, x):
aktiv [3]: +0,60 +0,20 0,00 // Baustein 3 schweigt hier

Schicht 2 summiert, jeder Baustein mit Höhe 0,5:
0,5·0,60 + 0,5·0,20 + 0,5·0,00 = 0,30 + 0,10 + 0 = 0,40
```

Und ohne den Knick? Dann geht der dritte Baustein mit −0,20 statt mit 0 in die Summe — und der ganze Stapel kollabiert wieder zu der Geraden vom Kapitelanfang:

```
ohne relu: 0,5·(+0,60) + 0,5·(+0,20) + 0,5·(−0,20) = 0,30
für jedes x: 0,5·(x+0,2) + 0,5·(x−0,2) + 0,5·(x−0,6) = 1,5·x − 0,3
```

Die Form bleibt dabei immer gleich: [3] rein, [3] raus — bei 128 Neuronen eben [128] rein und raus. In TensorFlow.js ist das eine Zeile oder ein einziges Wort:

```
const aktiv = roh.relu(); // [3] -> [3]
tf.layers.dense({ units: 3, activation: 'relu' }); // dasselbe im Layer
```

Neben ReLU begegnen dir noch zwei Klassiker — hier auf dieselben drei Zahlen angewendet:

roh	relu(x) max(0, x)	sigmoid(x) 0 … 1	tanh(x) −1 … +1
+0,60	0,60	0,65	0,54
+0,20	0,20	0,55	0,20
−0,20	0,00	0,45	−0,20

Sigmoid quetscht alles auf 0–1 — kennst du aus „Was ist ein Neuronales Netz?“, und in Kapitel 220 wird daraus dein Ja/Nein-Ausgang; **Tanh** auf −1 bis +1. In den versteckten Schichten hat sich ReLU durchgesetzt: billig zu rechnen, und der Gradient verhungert nicht so schnell.

Zurück zur Anwendung

Jedes `activation: 'relu'`, das dir in den Kapiteln bisher begegnet ist, war genau das: der Knick, der aus einem Geraden-Stapel einen Universal-Kurvenbauer macht. Ohne ihn wäre jede Bild-KI, jeder Chatbot und jedes Netz aus Kapitel 210 nur eine sehr teure Gerade.

Zum Weiterlesen

- **Das ReLU-Paper:** Vinod Nair & Geoffrey Hinton, *„Rectified Linear Units Improve Restricted Boltzmann Machines“* (ICML 2010, PDF) — cs.toronto.edu/~hinton/absps/reluICML.pdf — die Arbeit, die den Knick in den versteckten Schichten salonfähig gemacht hat.
- **Dein Kurven-Lego als Beweis:** Michael Nielsen, *Neural Networks and Deep Learning*, Kapitel 4 — neuralnetworksanddeeplearning.com/chap4.html — der visuelle Beweis, dass eine einzige versteckte Schicht jede Funktion annähern kann.
- **Warum ReLU tiefe Netze rettet:** Glorot, Bordes, Bengio, *„Deep Sparse Rectifier Neural Networks“* (AISTATS 2011) — proceedings.mlr.press/v15/glorot11a.html — der direkte Vergleich mit Sigmoid und Tanh, samt dem Argument des nicht verhungernden Gradienten.
- **Auf HerrKI:** 140 — Polynomiale Regression (die 8 Knicke im echten Training) und 220 — Binäre Klassifikation (wo Sigmoid als Ausgang zurückkommt)

Kurzer Check – mit Lösung

Lösungen für Lehrkräfte – für Lernende die Online-Version nutzen.

Frage 1: Was passiert, wenn man 10 Schichten OHNE Aktivierungsfunktion stapelt?

- Das Netz wird 10-mal schlauer.
- **✓ Mathematisch kollabiert alles zu einer einzigen Geraden — Tiefe bringt nichts.**
- Das Netz kann dann nur noch Bilder verarbeiten.

Linear nach linear bleibt linear — das hast du im Duell live gesehen: Der Schichten-Regler hat der linken Seite nicht geholfen.

Frage 2: Was rechnet ReLU?

- **✓ max(0, x) — Negatives wird zu null, Positives bleibt.**
- Es rundet jede Zahl auf ganze Werte.
- Es quadriert die Eingabe.

Nur ein Knick bei null — aber Summen verschobener Knicke können jede Kurve formen. Genau das war dein Kurven-Lego.

Frage 3: Warum kann ein Netz mit ReLU-Schichten beliebige Kurven lernen?

- Weil ReLU zufällige Werte erzeugt.
- **✓ Weil viele verschobene, skalierte Knicke sich zu jeder Form aufsummieren — und Training Position/Höhe jedes Knicks lernt.**
- Weil ReLU intern Polynome speichert.

Jedes Neuron liefert einen Baustein „Höhe × max(0, x − Position)“. 128 Neuronen = 128 Lego-Steine. Gradient Descent verschiebt sie, bis die Summe passt.

Das nimmst du mit

- **Gerade auf Gerade bleibt eine Gerade.** 10 Schichten ohne Aktivierung kollabieren mathematisch zu einer einzigen — Tiefe allein bringt exakt nichts.
- **ReLU ist max(0, x).** Negatives wird null, Positives geht durch. Ein Knick bei null, mehr nicht.
- **Aus Knicken lässt sich jede Kurve bauen.** Verschobene, skalierte Knicke aufsummiert — dein Kurven-Lego, nur mit 128 statt 3 Bausteinen.
- **Die Aktivierung wirkt elementweise.** [3] rein, [3] raus: jede Zahl einzeln durch den Knick, an der Shape ändert sich nichts.
- **ReLU hat die versteckten Schichten gewonnen.** Billig zu rechnen und der Gradient verhungert nicht so schnell — Sigmoid und Tanh sitzen heute vor allem am Ausgang.

So zitierst du dieses Kapitel

HerrKI: Warum eine Gerade nicht reicht (Aktivierungsfunktionen). herrki.de/145_warum_nichtlinear, ein privates Hobbyprojekt von Terragon.de, abgerufen am 09.09.2026.