Ein neuronales Netz in 150 Zeilen PHP — Brainy zum Anfassen

Kapitel online

Daniele Favis PHP-Bibliothek „Brainy" (157 Zeilen, MIT) als Lupe: vier Matrix-Operationen, forward() in fünf Zeilen, backPropagation() als Gradient Descent für ein ganzes Netz. Nach JavaScript übertragen und im Browser trainierbar — ein 2-3-1-Netz lernt XOR, mit Loss-Kurve, Entscheidungskarte und den Gewichten zum Zusehen.

Praxis · Kapitel 14 von 50 · ca. 20 Min. · 150 XP

Einfach gesagt: In rund 150 Zeilen PHP siehst du jede einzelne Zahl, mit der ein winziges Netz rechnet, während es „entweder-oder“ lernt.

TensorFlow.js ist ein Werkzeugkasten mit tausend Schubladen. Praktisch — aber wenn du `model.fit()` aufrufst, verschwindet die Rechnung hinter einer Klappe. Heute machen wir die Klappe auf: ein neuronales Netz aus **Schleifen und Arrays**, ohne Bibliothek, ohne GPU, ohne Magie. Die Vorlage dafür ist **„Brainy“** von **Daniele Favi** — eine PHP-Klasse mit **157 Zeilen**, in der du *jede* Zahl sehen kannst.

HerrKI-Tipp: Ein neuronales Netz ist **Matrixmultiplikation plus eine krumme Kurve** — alles andere ist Verpackung. Wenn du das einmal an 13 Zahlen gesehen hast, siehst du es bei 8 Milliarden Zahlen wieder.

Was du hier lernst

- Dass ein Netz aus **vier Matrix-Operationen** besteht — Transponieren, Punktprodukt, Addieren, elementweise Funktion
- Was `forward()` in fünf Zeilen tut: **$Z = f(W1^T \cdot x + b1)$**, dann noch einmal dasselbe
- Wie `backPropagation()` den Gradient Descent aus Kapitel 118 auf ein **ganzes Netz** anwendet
- Warum ein Netz mit **13 Zahlen** XOR lernen kann — und warum eines ohne versteckte Schicht daran scheitert
- **Warum Port und Original dieselben Zahlen liefern** — und was ein Framework obendrauflegt

Für den Unterricht

Einstieg (4 Min): Kapitel gemeinsam lesen, Alltagsbeispiel aufgreifen.

Erkunden am Gerät (12 Min): die interaktiven Module auf herrki.de/119_brainy_php ausprobieren.

Sichern mit Quiz (4 Min): „Kurzer Check" besprechen, „Das nimmst du mit" festhalten.

Warum ausgerechnet PHP?

PHP ist die Sprache eines sehr großen Teils des Webs: Sie läuft auf jedem Hosting-Paket, braucht kein Setup, keine Pakete, keine Umgebung. Genau deshalb hat Daniele Favi 2017 sein Lernprojekt **Brainy** darin geschrieben — „just for fun“ während seines KI-Studiums. Zwei Dateien:

Datei	Größe	Inhalt
`Matrix.php`	≈ 470 Zeilen	die Werkzeuge: transponieren, Punktprodukt, addieren, subtrahieren, elementweise Funktionen
`Brainy.php`	157 Zeilen	das Netz: `forward()`, `backPropagation()`, drei Ableitungen

Die **157 Zeilen** im Titel sind Brainy selbst — die Netz-Logik. Die Matrix-Klasse darunter ist reine Handarbeit für Zahlenlisten. Favi schreibt in seiner README einen Satz, den wir gern weitergeben, weil er ehrlich ist:

> *„If you are a web developer and you have just started studying AI, Brainy can be helpful to make the first steps in machine learning; **but there is no other good reason to use PHP for machine learning!**“*
> — Daniele Favi, README von github.com/danielefavi/brainy (MIT-Lizenz, © 2017 Daniele Favi)

Und was läuft hier auf der Seite? Kein PHP. HerrKI führt keine fremde Bibliothek auf dem Server aus — wir haben Brainys Rechenkern **nach JavaScript übertragen** (portiert) und lassen ihn in *deinem* Browser laufen. Die Methodennamen im Modul sind absichtlich dieselben wie im PHP-Original (`matrixDotProduct`, `backPropagation`, `w1`, `b1` …), damit die abgedruckten PHP-Ausschnitte und das, was du gleich anklickst, Zeile für Zeile zusammenpassen. Und ja: Wir haben nachgemessen. Port und Original liefern bei gleichen Startgewichten **dieselben Zahlen** (größte Abweichung nach 3000 Epochen: rund 10^{-17} — das ist die letzte Stelle einer Kommazahl).

Die Idee stammt aus einem eigenen Beitrag von 2026 auf news.terragon.de, der Favis Code interaktiv gemacht hat. Dort rechnete noch der Server mit; hier rechnet dein Browser.

Runter zum Tensor: Matrix mal Vektor von Hand

Bevor irgendetwas „lernt“, muss gerechnet werden — und das ist immer dasselbe: eine **Matrix mal einen Vektor**, das Punktprodukt aus „Tensor-Operationen“. So sieht es in Favis `Matrix.php` aus, gekürzt:

```
// Matrix.php, gekürzt – Daniele Favi, MIT-Lizenz
public function matrixDotProduct($m1, $m2) {
  // Passen die Formen? Spalten von A müssen Zeilen von B sein
  if ($cols_m1 != $rows_m2) throw new Exception('The matrices cannot be multiplied!');

  for ($i=0; $i<$rows_m1; $i++) { // jede Zeile von A …
    for ($j=0; $j<$cols_m2; $j++) { // … mal jede Spalte von B
      $prod[$i][$j] = 0;
      for ($k=0; $k<$rows_m2; $k++) {
        $prod[$i][$j] += $m1[$i][$k] * $m2[$k][$j]; // aufsummieren
      }
    }
  }
  return $prod;
}
```

Drei ineinander liegende Schleifen — das ist alles. **Keine Bibliothek der Welt macht etwas anderes**, sie macht es nur auf tausenden Kernen gleichzeitig. Dazu drei Werkzeuge:

- `matrixTranspose($m)` — kippt die Matrix: aus Zeilen werden Spalten. Shape [2,3] wird zu [3,2].
- `matrixSum($m1, $m2)` / `matrixSub($m1, $m2)` — elementweise addieren bzw. abziehen; beide Matrizen müssen exakt gleich groß sein.
- `matrixOperation('sigmoid', $m)` — wendet *eine Funktion auf jede einzelne Zahl* an. Genau hier kommt später die krumme Kurve ins Netz.

Am Gerät ausprobieren: herrki.de/119_brainy_php#abschnitt-3

Direkt zum Modul

Matrix mal Vektor — rechne mit

Links die Gewichtsmatrix **W** `[2,3]`, rechts der Eingabevektor **x** `[3,1]`. Ändere die Zahlen und lass eine Zeile ausrechnen — die beteiligten Felder werden markiert, darunter steht die Summe.

W `[2,3]`

2	0	-1
1	3	0.5

·

x `[3,1]`

1
2
4

= Ergebnis `[2,3] · [3,1] = [2,1]`

Zeile 1 rechnen | Zeile 2 rechnen | ↺ Zurücksetzen

▶ Ausgabe erscheint hier …

Merke: Das Ergebnis hat so viele *Zeilen* wie W und so viele *Spalten* wie x. Passt die mittlere Zahl nicht, wirft Brainy eine Ausnahme — auch das ist lehrreich: Ein Großteil aller KI-Fehlermeldungen sind Shape-Fehler.

Interaktives Modul – am Gerät ausprobieren

Vorwärts: forward() in fünf Zeilen

Jetzt bauen wir aus den Werkzeugen ein Netz. Unser Netz hat **zwei Eingänge, drei versteckte Neuronen und einen Ausgang** (kurz: 2-3-1), so wie in Kapitel 110. Der Vorwärtsdurchlauf ist zweimal dieselbe Rechnung:

> 1**Z = f(W1ᵀ · x + b1)** — Eingabe mal Gewichte, Bias drauf, krumme Kurve drüber. Ergebnis: die versteckte Schicht [3,1]
>
> 2**A = g(W2ᵀ · Z + b2)** — dasselbe noch einmal, jetzt von der versteckten Schicht zum Ausgang [1,1]

Und so steht es im Original — links das PHP von Daniele Favi, rechts unser JavaScript-Port. Die Mathematik ist sprachunabhängig; nur die Klammern wandern:

```php
// Brainy.php (gekürzt) – Favi, MIT
$dot = $this->matrixDotProduct(
         $this->matrixTranspose($w1), $input);
$sum = $this->matrixSum($dot, $b1);
$z = $this->matrixOperation(
         $this->activation_fun, $sum);

$dot = $this->matrixDotProduct(
         $this->matrixTranspose($w2), $z);
$sum = $this->matrixSum($dot, $b2);
$a = $this->matrixSoftmax($sum);
```

```javascript
// modul.js – 1:1 portiert
var dot = this.matrixDotProduct(
        this.matrixTranspose(w1), input);
var sum = this.matrixSum(dot, b1);
var z = this.matrixOperation(
        this.activation_fun, sum);

dot = this.matrixDotProduct(
        this.matrixTranspose(w2), z);
sum = this.matrixSum(dot, b2);
var a = this.matrixSoftmax(sum);
```

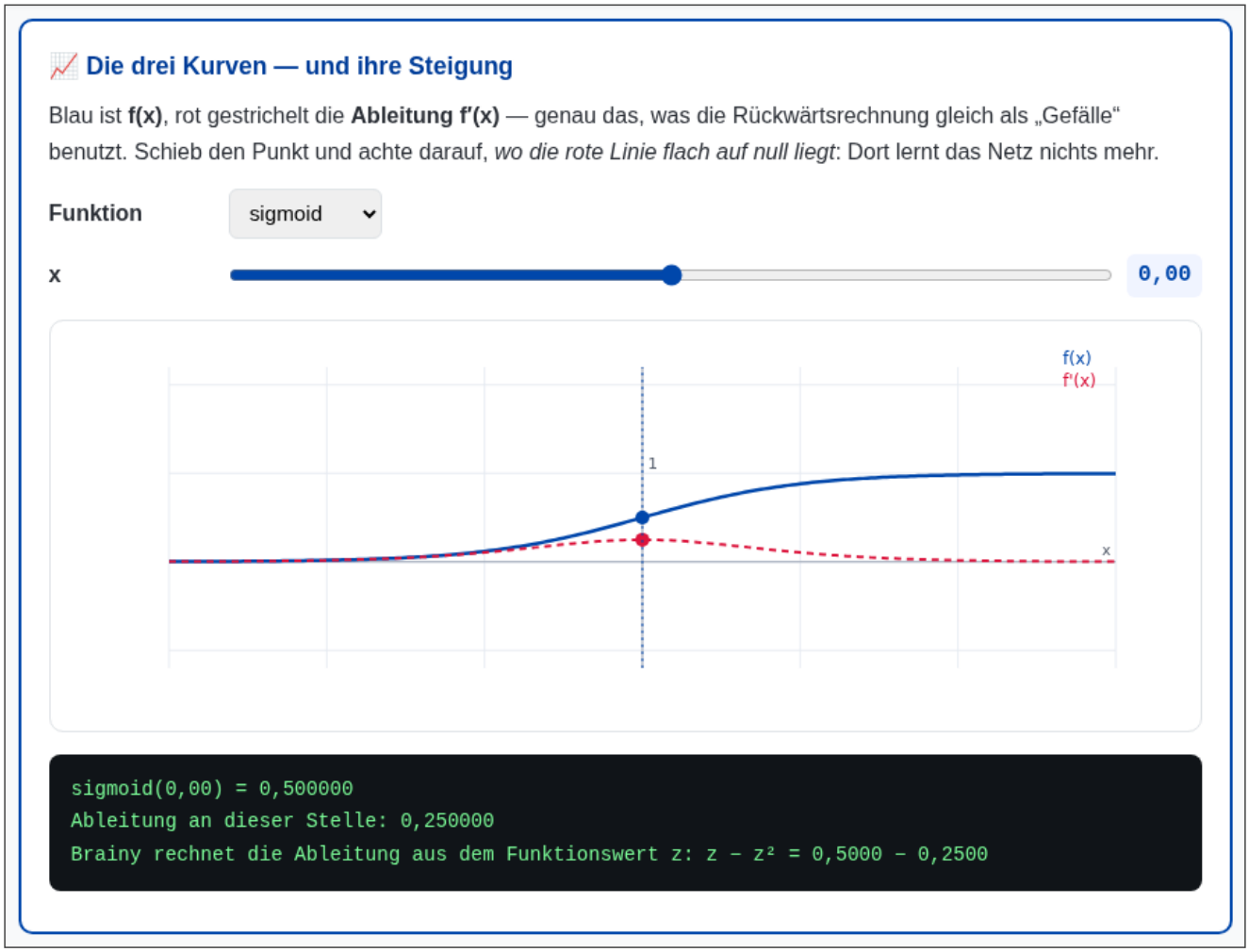
Die „krumme Kurve“ `f` heißt **Aktivierungsfunktion** (Kapitel 145 geht ihr auf den Grund). Brainy kennt genau drei — und rechnet ihre Ableitung, die der Gradient Descent braucht, aus dem *Ergebnis* z statt aus der Eingabe x. Das spart einen kompletten Rechenschritt:

Funktion	Formel	Ableitung in Brainy
`relu`	x > 0 ? x : 0	z > 0 ? 1 : 0
`sigmoid`	$1 / (1 + e^{-x})$	$z - z^2$ (= z·(1−z))

`tanh`	tanh(x)	$1 - z^2$

Am Gerät ausprobieren: herrki.de/119_brainy_php#abschnitt-4

Direkt zum Modul



Interaktives Modul – am Gerät ausprobieren

Rückwärts: backPropagation() — Gradient Descent für ein ganzes Netz

In Kapitel 118 hast du eine Kugel ins Tal geschickt: Gefälle ertasten, einen Schritt machen, wiederholen. Genau das passiert hier — nur für **13 Gewichte gleichzeitig** (6 in W1, 3 in b1, 3 in W2, 1 in b2). Der Weg zurück geht in vier Schritten, und man kann sie in Worten sagen:

1**Fehler am Ausgang:** `diff = A − Soll`. Mehr nicht — das ist die Ableitung des Loss (Kapitel 115) nach der Ausgabe.

2**Gewichte der letzten Schicht korrigieren:** `W2 = W2 − lernrate · (diff · Zᵀ)ᵀ` — genau die Zeile aus 118, nur mit Matrizen.

3**Fehler durchreichen (Kettenregel):** `dZ = (W2 · diff) ⊙ f′(Z)`. Der Fehler wird rückwärts durch die Gewichte geschoben und mit der *Steigung der Aktivierung* multipliziert — dem roten Strich aus dem Modul oben. Ist die Steigung dort null, kommt hinten null an: Das Neuron lernt nicht mehr.

4**Erste Schicht korrigieren:** `W1 = W1 − lernrate · (dZ · xᵀ)ᵀ` — dieselbe Zeile wie in Schritt 2, eine Schicht weiter vorn.

```
// Brainy.php, backPropagation() – stark gekürzt (Favi, MIT)
$diff = $this->matrixSub($r['A'], $output); // 1) Fehler am Ausgang
$prod = $this->matrixDotProduct($diff, $this->matrixTranspose($r['Z']));
$corr_mat = $this->matrixTimesValue($prod, $this->learning_rate);
$w2 = $this->matrixSub($w2, $this->matrixTranspose($corr_mat)); // 2) W2 korrigieren

if ($this->activation_fun == 'sigmoid') $z_der = $this->sigmoidDerivative($r['Z']);
$prod = $this->matrixDotProduct($w2, $diff);
$dZ = $this->matrixProductValueByValue($prod, $z_der); // 3) Kettenregel

$prod = $this->matrixDotProduct($dZ, $this->matrixTranspose($input));
$corr_mat = $this->matrixTimesValue($prod, $this->learning_rate);
$w1 = $this->matrixSub($w1, $this->matrixTranspose($corr_mat)); // 4) W1 korrigieren
```

Zwei Dinge lohnen den zweiten Blick. Erstens: `learning_rate` steht in beiden Korrekturen an derselben Stelle — sie ist die **Schrittweite**, sonst nichts. Zweitens: Die Methode heißt `backPropagation($r, $input, $output, $w1, $w2, $b1, $b2)`, während `forward($input, $w1, $b1, $w2, $b2)` die Gewichte in *anderer* Reihenfolge nimmt. Wer das verwechselt, trainiert stundenlang Unsinn ohne eine einzige Fehlermeldung. Unser Port hat den Dreher **absichtlich übernommen** — er gehört zum Original.

Ein Beispiel, ein Update. Brainy korrigiert die Gewichte nach *jedem einzelnen* Trainingsbeispiel (stochastisch), nicht nach einem ganzen Stapel. Ein „Batch“, wie ihn TensorFlow standardmäßig bildet, kommt hier nicht vor. Deshalb ruckelt die Loss-Kurve gleich auch sichtbar, statt glatt zu fallen.

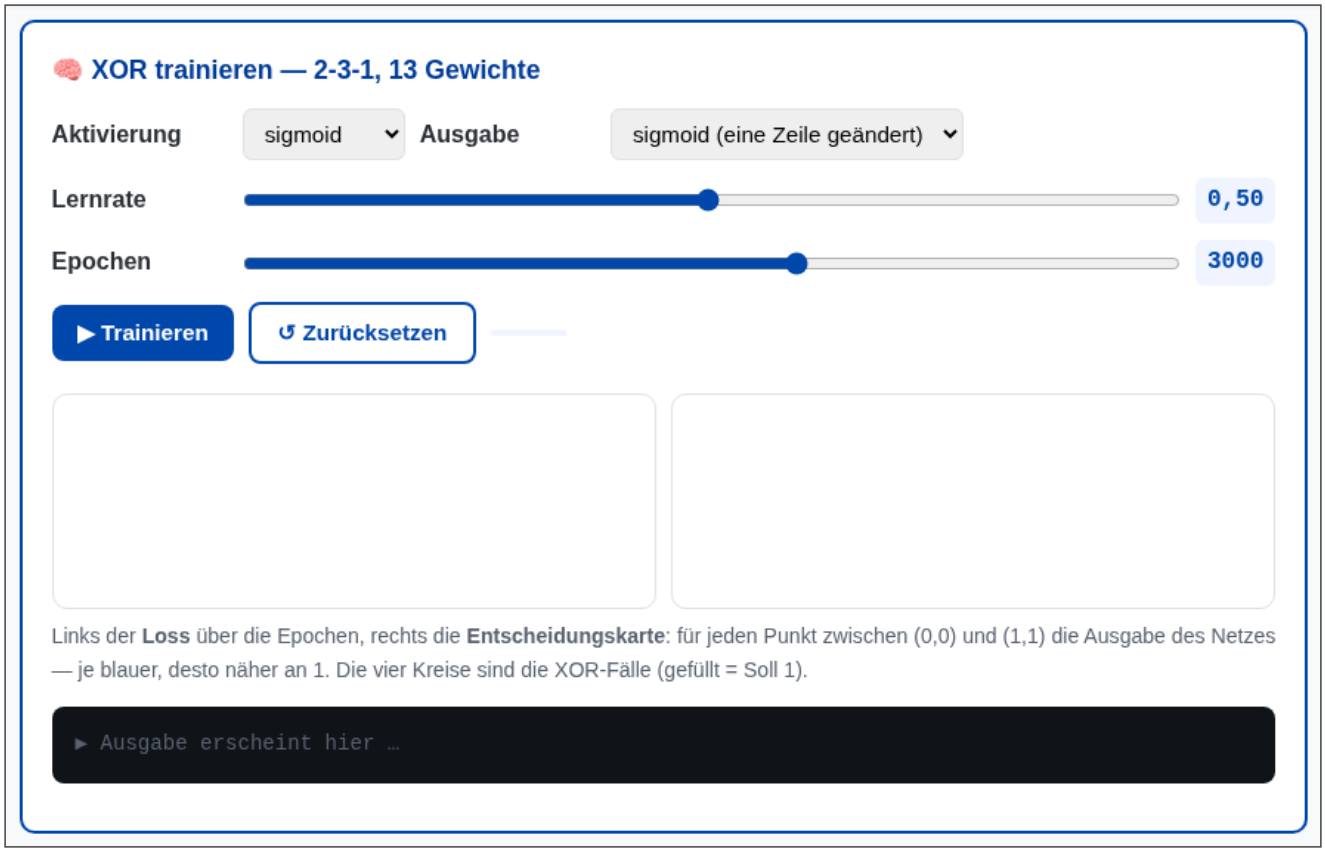
Zum Anfassen: XOR lernen — live in deinem Browser

XOR („entweder oder“) ist die klassische Prüfung für ein Netz: 0/0 → 0, 0/1 → 1, 1/0 → 1, 1/1 → 0. Man kann diese vier Punkte **nicht mit einer geraden Linie trennen** — ein Netz ohne versteckte Schicht scheitert daran garantiert. Mit drei versteckten Neuronen und einer krummen Kurve geht es. Die Startgewichte sind fest (sie stammen aus Favis eigener Beispieldatei), damit dein Ergebnis reproduzierbar ist.

> **Ein Wort zum Umschalter „Ausgabe“.** Brainy gibt die letzte Schicht durch `matrixSoftmax()` aus — bei mehreren Ausgabeneuronen genau richtig, denn Softmax verteilt die Wahrscheinlichkeit über die Klassen. Unser Netz hat nur eines, und Softmax über eine einzige Zahl ergibt immer 1. Für XOR nutzen wir deshalb an dieser einen Stelle `sigmoid`; mit dem Umschalter siehst du den Unterschied.

Am Gerät ausprobieren: herrki.de/119_brainy_php#abschnitt-6

Direkt zum Modul



Interaktives Modul – am Gerät ausprobieren

Probier das aus:

- **Lernrate ganz nach links (0,01):** Die Kurve hängt fast waagerecht — 3000 Epochen reichen nicht. Kein Defekt, sondern Kapitel 118.
- **Lernrate 1 mit `relu`:** Die Ausgabe wird `NaN` — genau das „Loss: NaN“ aus Kapitel 118

: Zu große Schritte lassen die Gewichte explodieren.

- **Ausgabe auf „matrixSoftmax (Original)“:** Die vier Ausgaben sind dann *exakt* 1 oder *exakt* 0, nie etwas dazwischen, die Entscheidungskarte eine einfarbige Fläche.

Zurück zur Anwendung: Wann reicht PHP, wann braucht es TensorFlow?

Unser Netz hat **13 Gewichte**. Ein Sprachmodell der 8-Milliarden-Klasse hat **rund 615 Millionen mal so viele**. Beide rechnen dieselbe Mathematik — Punktprodukt, Bias, krumme Kurve, Fehler zurück. Was ein Framework hinzufügt, sind drei Dinge, die genau bei diesem Größensprung anfangen zu zählen:

Was das Framework bringt	Warum es bei 13 Gewichten egal ist
Autograd — die Ableitungen werden automatisch hergeleitet	Hier stehen sie von Hand im Code: drei Ableitungen, fertig. Bei 200 Schichten schreibt die niemand mehr selbst.
GPU — tausende Rechenkerne rechnen ein Punktprodukt gleichzeitig	Unsere drei Schleifen reichen für 4 Beispiele. Für 4 Millionen Bilder bräuchten sie Jahre (mehr dazu in Kapitel 390).
Batches — viele Beispiele in einem Rutsch, ein Update	Brainy macht ein Update pro Beispiel. Bei Millionen Beispielen wäre das unbezahlbar langsam und obendrein unruhig.

Gleich bleibt der Kern: `W·x + b`, eine krumme Kurve, Fehler messen, Gefälle ertasten, Schritt machen. Wenn du im nächsten Kapitel zum ersten Mal `model.fit()` aufrufst, weißt du, was hinter der Klappe steckt — du hast es gesehen.

Was heißt das für dich?

Zwei Dinge nimmst du mit, die weit über PHP hinausgehen. Erstens: **KI ist lesbar.** Wer Punktprodukt, Bias und Ableitung verstanden hat, kann jede Netz-Implementierung lesen — die Namen wechseln, die Rechnung nicht.

Zweitens: **kleine, lesbare Projekte sind die besten Lehrer.** Daniele Favi hat sein Lernprojekt großzügig unter MIT-Lizenz geteilt — 157 Zeilen, an einem Abend zu lesen, ohne Installation, ohne Framework. Genau das macht so ein Projekt wertvoll: Man kann es *ganz* durchlesen, statt es nur zu benutzen. Und wer einmal jede Zeile gelesen hat, glaubt Code nicht mehr nur — er versteht ihn. Danke dafür, Daniele.

Zum Weiterlesen

- **Der Originalcode:** Daniele Favi, *Brainy — a simple PHP class for machine learning* (MIT-Lizenz, © 2017) — github.com/danielefavi/brainy — zwei Dateien, an einem Abend lesbar. **Danke, Daniele.**
- **Favis eigene Erklärung:** danielefavi.com/brainy-a-simple-php-class-for-machine-learning — wie er sein Netz aufgebaut hat, mit seinen eigenen Worten.
- **Die Mathematik sauber hergeleitet:** Michael Nielsen, *Neural Networks and Deep Learning*, Kapitel 2 — neuralnetworksanddeeplearning.com/chap2.html — dieselben vier Schritte, nur mit Beweis.
- **Ursprung dieses Kapitels:** ein eigener Beitrag von 2026 auf news.terragon.de, der Favis Code interaktiv gemacht hat. Die Seite ist inzwischen offline; die Idee lebt hier weiter.
- **Auf HerrKI:** 060 — Tensor-Operationen (das Punktprodukt), 110 — Was ist ein neuronales Netz?, 115 — Wie falsch ist falsch?, 118 — Gradient Descent und 145 — Warum es krumm sein muss.

Kurzer Check – mit Lösung

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

Frage 1: Du multiplizierst eine Matrix [2,3] mit einem Vektor [3,1]. Welche Form hat das Ergebnis?

- [3,3] — die größere Zahl gewinnt.
- **✔ [2,1] — so viele Zeilen wie die Matrix, so viele Spalten wie der Vektor.**
- [3,2] — Zeilen und Spalten werden getauscht.

Die inneren Zahlen (3 und 3) müssen zusammenpassen und fallen weg, die äußeren bleiben stehen: [2,3]·[3,1] = [2,1]. Passen die inneren nicht, wirft Brainy „The matrices cannot be multiplied!“.

Frage 2: Was macht die learning_rate in backPropagation()?

- Sie bestimmt, wie viele Neuronen die versteckte Schicht hat.
- Sie legt fest, wie oft alle Trainingsbeispiele wiederholt werden.
- **✔ Sie ist die Schrittweite: Der berechnete Korrekturwert wird mit ihr multipliziert, bevor er von den Gewichten abgezogen wird.**

Im Code steht sie an genau zwei Stellen — einmal für W2, einmal für W1 — jeweils als matrixTimesValue($prod, $this->learning_rate). Die Anzahl der Neuronen und die Epochen sind davon völlig unabhängig.

Frage 3: Du stellst die Lernrate auf den Höchstwert und trainierst mit relu. Was passiert am wahrscheinlichsten?

- **✔ Die Schritte werden zu groß, die Korrekturen schießen übers Ziel — der Loss springt oder bleibt oben hängen.**
- Das Netz lernt genau dreimal so schnell und ist früher fertig.
- Nichts — die Lernrate wirkt erst ab 10.000 Epochen.

Dasselbe wie bei der Kugel in Kapitel 118: zu große Schritte springen über das Tal. Bei relu kommt hinzu, dass ein Neuron, das einmal ins Negative gerutscht ist, die Ableitung 0 hat und nie zurückkommt — es ist tot. Erste Hilfe bleibt immer: Lernrate durch 10 teilen.

Das nimmst du mit

- **Ein Netz sind vier Matrix-Operationen.** Transponieren, Punktprodukt, Addieren, eine Funktion auf jedes Element — mehr braucht `forward()` nicht.
- **Vorwärts ist zweimal dieselbe Zeile.** $Z = f(W1^T \cdot x + b1)$, dann $A = g(W2^T \cdot Z + b2)$. Jede weitere Schicht ist eine Wiederholung.
- **Rückwärts ist Kapitel 118 mit Matrizen.** Fehler am Ausgang, Kettenregel durch die Gewichte, Ableitung der Aktivierung dazu, Schrittweite mal Korrektur — fertig.
- **Die Sprache ist egal.** PHP-Original und JavaScript-Port liefern bei gleichen Startgewichten dieselben Zahlen bis zur letzten Kommastelle. Was ein Framework hinzufügt, sind Autograd, GPU und Batches — nicht andere Mathematik.
- **Kleine, lesbare Projekte sind der beste Einstieg** — danke, Daniele Favi.

Genug Handarbeit. Im nächsten Kapitel, „Einfache Lineare Regression“, überlässt du dieselbe Rechnung zum ersten Mal einem Framework: TensorFlow.js, `model.fit()`, drei Zeilen. Der Unterschied ist nur — du weißt jetzt, was drin steckt.

So zitierst du dieses Kapitel

HerrKI: Ein neuronales Netz in 150 Zeilen PHP — Brainy zum Anfassen. herrki.de/119_brainy_php, ein privates Hobbyprojekt von Terragon.de, abgerufen am 09.09.2026.