📘 KI-ENNA: CODE-TUTORIAL

Dies ist *(E)in (N)euronales (N)etz zum (A)usprobieren*, das auf einem Raspberry Pi Pico (Hardware) oder Thonny (Software) läuft. Es verdeutlicht die Mathematik hinter neuronalen Netzen, ohne auf externe Bibliotheken, teure Hardware oder Cloud-Services angewiesen zu sein.

🔧 1. Vorbereitungen

```
import math
```

- Ermöglicht mathematische Funktionen wie bspw. `exp`.

🧠 2. Aktivierungsfunktionen

```
def relu(x): …
def leaky_relu(x, alpha=0.01): …
def tanh(x): …
def sigmoid(x): …
def softmax(x): …
```

- Aktivierungsfunktionen sind für das Lernverhalten neuronaler Netze entscheidend.
- Mit ihnen können insbesondere komplexe Muster und Strukturen verarbeitet werden.
- Jede Aktivierungsfunktion nimmt eine Liste `x` entgegen und gibt die aktivierten Werte zurück.
- Bereits vorprogammierte Aktivierungsfunktionen: `relu`, `leaky_relu`, `tanh`, `sigmoid` und `softmax`.

🧮 3. Neuronen

```
def neuron(x, w, b, activation): …
```

- Neuronen sind für die Weitergabe von Werten an andere Neuronen verantwortlich.
- Sie werden über die Aktivierungsfunktionen aktiviert oder bleiben deaktiviert.
- In MicroPython lässt sich ein einzelnes Neuron wie folgt simulieren:
  - Multiplikation der Eingabewerte `x` mit Gewichten `w` unter Addition der Biase `b`
  - Aktivierung über `relu`, `leaky_relu`, `tanh`, `sigmoid` oder `softmax`.
  - Rückgabe: Liste aktivierter Werte

📐 4. Mathematische Hilfsfunktionen

```
def zero_dim(x): …
def add_dim(x, y): …
def zeros(rows, cols): …
def transpose(M): …
def print_matrix(M, decimals=3): …
```

- Neuronale Netze benötigen für die Datenverarbeitung mathematische Hilfsfunktionen.
- Die mathematischen Hilfsfunktionen sollten in MicroPython nicht verändert werden!
- Folgende mathematische Hilfsfunktionen sind bereits vorprogrammiert:
  - Null-Initialisierung
  - Addition von Vektoren
  - Transponierung von Matrizen
  - Vordefinierte Form der Matrix-Ausgabe

🏗️ 5. Dichte eines neuronalen Netzes

```
def dense(nunit, x, w, b, activation): …
```

- Neuronale Netze basieren auf mehreren Schichten.
- Die Funktion bildet eine Schicht mit `nunit` Neuronen ab.
- Jedes Neuron verarbeitet den Input `x` mit individuellen Gewichten und Biaswerten.
- Die Anzahl der Neuronen und Schichten bestimmt dabei die Dichte eines neuronalen Netzes.

📥 6. Einbindung der Gewichte (w) und Bias (b)

```
W1 = …
b1 = …
```

- Das Training neuronaler Netze erfordert bspw. TensorFlow in Python.
- Gewicht `w` und Bias `b` der vortrainierten Modelle können in MicroPython überführt werden.

🔄 7. Transponierung der Gewichte

```
w1 = transpose(w1)
…
```

- TensorFlow in Python und MicroPython nutzen unterschiedliche Speicherformate.
- Die Transponierung ermöglicht die Datenverarbeitung in MicroPython.

🧪 8. Datensatz

```
Xtest = […]
ytrue = […]
```

- Standardisierte (z-Transformation) Eingabedaten `Xtest` als unabhängige Variablen.
- Tatsächliche Werte `ytrue` der abhängigen Variable zum Abgleich mit den Ausgabedaten.

🧮 9. Neuronales Netz

```
yout1 = dense(2, transpose(Xtest), w1, b1, 'relu')
yout2 = dense(3, yout1, w2, b2, 'sigmoid')
yout3 = dense(2, yout2, w3, b3, 'relu')
ypred = dense(1, yout3, w4, b4, 'sigmoid')
```

- Der Datensatz wird Schicht für Schicht durch das neuronale Netz geleitet.
- Die Eingabeschicht besteht bspw. aus 2 Neuronen mit `w1` und `b1` auf Basis von `relu`.
- Gewicht `w` und Bias `b` müssen immer (!) passend (siehe 6. Schritt) gewählt werden.

📊 10. Konvertierung in binäre Klassen

```
ypred_class = [1 if i > 0.5 else 0 for i in ypred[0]]
```

- Die Ausgabedaten ermöglichen kriteriengeleitet eine binäre Klassifikation (1 = pos. / 0 = neg.).
- Diese werden im nachfolgenden Schritt mit den tatsächlichen Werte `ytrue` abgeglichen.

📋 11. Auswertung mit Confusion Matrix

```
def classification_report(ytrue, ypred): …
```

- Damit werden die Ausgabedaten `ypred_class` mit den tatsächlichen Werten `ytrue` abgeglichen.
- Auf dieser Grundlage lässt sich die Accuracy als Anteil richtiger Vorhersagen berechnen:
  - True Positives (TP) oder False Positives (FP)
  - True Negatives (TN) oder False Negatives (FN)

🧪 12. Ergebnisse anzeigen

```
print(ypred_class)
print(classification_report(ytrue, ypred_class))
```

- Schließlich müssen die Ergebnisse des neuronalen Netzes nur noch ausgegeben werden.

🧑‍🔬 13. Flexibilität und Kontrolle

- Die Neuronenanzahl kann flexibel angepasst werden, bspw. von `dense(2, ...)` zu `dense(4, …)`.
- Entsprechend kann die Anzahl an Schichten über eine weitere `dense`-Zeile angepasst werden.
- Achtung: Die Anzahl an Neuronen und Schichten muss mit dem 6. Schritt deckungsleich sein!
- Hierfür können Gewichte `w` und Biase `b` flexibel eingetragen werden (Finetuning).
- Alternativ sind diese aus vortrainierten Modellen (TensorFlow in Python) zu entnehmen.
- Dabei können auch neue Datensätze vortrainiert und in MicroPython überführt werden.
- Aktivierungsfunktionen können ebenfalls flexibel ausprobiert werden, bspw. `relu` statt `tanh`.