Dokumentation

Programmentwurf Algorithmen und Verfahren

Studiengang Informatik (TIK 24)
an der Dualen Hochschule Baden-Württemberg Ravensburg Standort Friedrichshafen

von

Omer Butt

Julian Greiner

Timo Johannsen

Benjamin Peiter

Abgabedatum: 27.04.2025

Kurs: Algorithmen und Verfahren

Ziel des Programms

Das Ziel des Programms ist es eine DNF (disjunktive Normalform) mithilfe eins zweischichtigen, vorwärtsgetriebenen neuronalen Netzes zu realisieren.

Aufbau des Netzes

Das Netz besteht aus drei Schichten:

Eingabeschicht (Input Layer)

Die Eingabeschicht besteht aus n Eingabewerten $x \in \{-1, 1\}$, wobei -1 dem logischen Falsch und 1 dem logischen Wahr entspricht.

Zwischenschicht (Hidden Layer)

Die Zwischenschicht besteht aus m Neuronen $z_1, \dots, z_m$ wobei jedes z das Ergebnis eines Monoms darstellt.

Ausgabeschicht (Output Layer)

Die Ausgabeschicht besitzt nur ein Neuron y, welches das Ergebnis der DNF mit gegebenem Input repräsentiert.

Technische Umsetzung

Signum Funktion

```
# Signum-Funktion (sig), die die Werte eines Vektors auf -1 oder 1
setzt
def sig(x):
    """
    Wendet die Signum-Funktion auf jeden Wert eines Vektors an.

    Parameter:
    - x: Eingabevektor (numpy-Array)

    Rückgabe:
    - Ein numpy-Array mit Werten -1 oder 1.
    """
    ret = np.copy(x)  # Kopiere den Eingabevektor
    for i in range(np.shape(x)[0]):  # Iteriere über alle Elemente
        if x[i] >= 0:
            ret[i] = 1  # Setze positive Werte auf 1
        else:
            ret[i] = -1  # Setze negative Werte auf -1
    return ret.astype(np.int64)  # Rückgabe als Integer-Array
```

Als Aktivierungsfunktion wird die Signum-Funktion genutzt. Hier bekommt diese einen Eingabevektor und wendet auf jedes Element des Vektors die Signumfunktion an.

Hilfsfunktion zur Berechnung der genutzten Literale

```
# Funktion zur Berechnung der Anzahl der verwendeten Literale in einem
Monom
def get_used_literals(monome):
    """
    Zählt die Anzahl der nicht-null Werte in einem Monom.

    Parameter:
    - monome: Ein Array, das ein Monom repräsentiert

    Rückgabe:
    - Anzahl der verwendeten Literale (Werte ungleich 0).
    """
    literal_count = 0
    for elem in monome:
        if elem != 0:
            literal_count += 1  # Erhöhe den Zähler für jedes
nicht-null Element
    return literal_count
```

Hierbei wird ein Monom übergeben und die Anzahl der genutzten Literale errechnet. Diese brauchen wir später, um den Schwellenwert für die Zwischenschicht zu bekommen, damit wird die logische UND-Funktion realisieren können.

Berechnung des Schwellenwerts der Zwischenschicht

```
# Berechnung der Schwellenwerte für das Z-Layer
def calculate_treshold_z(w):
    """
    Berechnet die Schwellenwerte (v) für das Z-Layer basierend auf der
Anzahl der verwendeten Literale.

    Parameter:
    - w: Gewichtsmatrix (numpy-Array)

    Rückgabe:
    - Ein numpy-Array mit Schwellenwerten.
    """
    v = np.empty(w.shape[0], dtype=np.float64)  # Initialisiere den
Schwellenwert-Vektor
    for i in range(w.shape[0]):
        v[i] = get_used_literals(w[i]) - 0.5  # Berechnung des
Schwellenwerts
    return v
```

Hier werden die Schwellenwerte für die Zwischenschicht errechnet. Jedes Monom wird auf die Anzahl der genutzten Variablen überprüft. Diese Anzahl wird daraufhin -0,5 gerechnet. Das ermöglicht es die logische UND-Funktion zu realisieren.

Berechnung Zwischenschicht (z-Layer)

```
# Berechnung des Z-Layers
def calculate_z(w, input, v):
    """
    Berechnet die Werte des Z-Layers.

    Parameter:
    - w: Gewichtsmatrix (numpy-Array)
    - input: Eingabevektor (numpy-Array)
    - v: Schwellenwerte (numpy-Array)

    Rückgabe:
    - Ein numpy-Array mit den berechneten Z-Werten.
    """
    z = np.empty(w.shape[0], dtype=np.float64)  # Initialisiere Z
    for j in range(w.shape[0]):
        z[j] = np.dot(w[j], input) - v[j]  # Skalarprodukt minus
Schwellenwert
    z = sig(z)  # Wende die Signum-Funktion auf Z an
    return z
```

Mithilfe der calculate_z-Funktion ist es möglich die logische UND-Funktion zu realisieren. Als Parameter übergeben wir die DNF als Gewichtsmatrix, die Eingabevariablen, sowie die Schwellenwerte, die wir vorher mithilfe der calculate_treshold-Funktion errechnet haben. Zur errechnung aller z wird folgende Formel verwendet:

$$\underline{z} = \operatorname{sgn}(w \cdot \underline{x} - \underline{v})$$

Statt der Matrixmultiplikation, wird eine Schleife und das Skalarprodukt verwendet, um alle z_j zu berechnen, was zum gleichen Ergebnis führt.

Berechnung der Schwellenwerte und Gewichte für die Ausgabeschicht (y-Layer)

```
def calculate_treshold_and_weights_y(z):
    """
    Berechnet die Schwellenwerte und Gewichte für das Y-Layer.

    Parameter:
    - z: Anzahl der Z-Werte (int)

    Rückgabe:
    - W: Gewichtsmatrix (numpy-Array)
    - V: Schwellenwerte (numpy-Array)
    """
    W = np.ones((z), np.float64)  # Initialisiere die Gewichtsmatrix
mit Einsen
    V = -(z - 1)  # Berechnung der Schwellenwerte
    return W, V
```

Die Funktion Berechnet die Schwellenwerte und Gewichte für die Ausgabeschicht. Dazu wird die Anzahl der Monome, bzw. Anzahl der Elemente im z-Vektor benötigt. Um später die logische ODER-Funktion zu realisieren, füllen wir die Gewichtsmatrix W mit Einsen. Der Schwellenwert V ergibt sich dann aus $V = -(Anzahl\, z - 1)$

Berechnung Ausgabeschicht (y-Layer)

```
# Berechnung des Y-Layers
def calculate_y(W, V, z):
    """
    Berechnet die Ausgabe des Y-Layers.

    Parameter:
    - W: Gewichtsmatrix (numpy-Array)
    - V: Schwellenwerte (numpy-Array)
    - z: Eingabevektor für das Y-Layer (numpy-Array)

    Rückgabe:
    - y: Ausgabe des Y-Layers (int)
    """
    y = np.matmul(W, z.T)   # Matrixmultiplikation
    y = y - V  # Subtrahiere die Schwellenwerte
    if y >= 0:
        y = 1  # Setze positive Werte auf 1
    else:
        y = -1  # Setze negative Werte auf -1
    return y
```

Diese Funktion berechnet die Ausgabeschicht, und somit das Ergebnis unseres Durchlaufs. Die Berechnung basiert auf folgender Formel:

$$y = \mathrm{sgn}(W \cdot \underline{z} - V)$$

Durch die gewählten Gewichte und Schwellenwerte ist es möglich mithilfe der Aktivierungsfunktion hier die logische ODER-Funktion umzusetzen, wodurch die DNF nun komplett realisiert ist.

Fehlerrückübertragung (Backpropagation)

```
# Funktion zur Berechnung der Deltas für die Backpropagation
def calculate_deltas(input, p, lernrate, z, y, w, v, W, V):
    """
    Berechnet die Deltas für die Anpassung der Gewichte und
Schwellenwerte während der Backpropagation.

    Parameter:
    - input: Eingabevektor (numpy-Array)
    - p: Erwartete Ausgabe (int)
    - lernrate: Lernrate (float)
    - z: Z-Werte (numpy-Array)
    - y: Ausgabe des Y-Layers (int)
    - w: Gewichtsmatrix für das Z-Layer (numpy-Array)
    - v: Schwellenwerte für das Z-Layer (numpy-Array)
    - W: Gewichtsmatrix für das Y-Layer (numpy-Array)
    - V: Schwellenwerte für das Y-Layer (numpy-Array)

    Rückgabe:
    - delta_w, delta_v, delta_W, delta_V: Anpassungen für die Gewichte
und Schwellenwerte
    """
    delta_w = np.copy(w)  # Kopiere die Gewichtsmatrix
    delta_v = np.copy(v)  # Kopiere die Schwellenwerte
    delta_W = np.copy(W)  # Kopiere die Gewichtsmatrix für das Y-Layer
    delta_V = np.copy(V)  # Kopiere die Schwellenwerte für das Y-Layer

    error = p - y  # Berechne den Fehler

    for j in range(W.size):
        delta_W[j] = lernrate * error * z[j]  # Anpassung der Gewichte
im Y-Layer
        delta_v[j] = -lernrate * error * W[j]  # Anpassung der
Schwellenwerte im Z-Layer
        for k in range(input.size):
            delta_w[j][k] = lernrate * W[j] * error * input[k]  #
Anpassung der Gewichte im Z-Layer
    delta_V = -lernrate * error  # Anpassung der Schwellenwerte im
Y-Layer

    return delta_w, delta_v, delta_W, delta_V
```

Um die Fehlerrückübertragung zu realisieren werden durch diese Funktion die Deltas nach folgenden Formeln berechnet:

$$\Delta W_{1j} = \eta \cdot (p - y) \cdot z_j \qquad \Delta V = -\eta \cdot (p - y)$$

$$\Delta w_{jk} = \eta \cdot W_{1j}(p - y) \cdot x_k \qquad \Delta v_j = -\eta \cdot (p - y) \cdot W_{1j}$$

Hierzu werden Trainingsdaten benötigt, welche die Eingabevariablen enthalten, sowie den erwarteten Output. Wenn die tatsächliche Ausgabe nicht der erwarteten entspricht, ergibt sich dadurch für jeden Schwellenwert und jedes alles Gewichte ein Delta, was mit den Ursprungswerten verrechnet wird um das Netz zu trainieren.

Trainieren des Netzes

```
def feedforward_backpropagation(dnf, trainings_data, lerningsrate,
epochs, mode):
    """
    Führt den Feedforward- und Backpropagation-Algorithmus aus.

    Parameter:
    - dnf: Gewichtsmatrix (numpy-Array)
    - trainings_data: Trainingsdaten (Liste von Tupeln)
    - lerningsrate: Lernrate (float)
    - epochs: Anzahl der Epochen (int)
    - mode: Modus zur Veränderung der Gewichte (0 = keine Veränderung,
1 = leichte Veränderung, 2 = zufällige Werte)
    """
    match mode:
        case 0:
            w = dnf  # Keine Veränderung der Gewichte
        case 1:
            w = change_weights(dnf, False)  # Leichte Veränderung der
Gewichte
        case 2:
            w = change_weights(dnf, True)  # Zufällige Gewichte

    v = calculate_treshold_z(w)  # Berechnung der Schwellenwerte für
das Z-Layer
    y_treshhold = calculate_treshold_and_weights_y(w.shape[0])  #
Berechnung der Schwellenwerte und Gewichte für das Y-Layer
    W, V = y_treshhold

    for epoch in range(epochs):
        random_int = random.randint(0, len(trainings_data) - 1)  #
Zufällige Auswahl eines Trainingsdatensatzes
        input = trainings_data[random_int][0]  # Eingabevektor
        p = trainings_data[random_int][1]  # Erwartete Ausgabe
        z = calculate_z(w, input, v)  # Berechnung der Z-Werte
        y = calculate_y(W, V, z)  # Berechnung der Ausgabe

        # Ausgabe der aktuellen Epoche und des Fehlers
        print(f"Epoche: {epoch + 1:03d} | Fehler: {p - y:3d} | y:
{y:2d} | p: {p:2d}")
```

```
        # Berechnung der Deltas und Anpassung der Gewichte und
Schwellenwerte
        delta_w, delta_v, delta_W, delta_V = calculate_deltas(input, p,
lerningsrate, z, y, w, v, W, V)
        w = np.add(w, delta_w)
        v = np.add(v, delta_v)
        W = np.add(W, delta_W)
        V = np.add(V, delta_V)
```

Um das Trainieren des Netzes zu starten, wird diese Funktion verwendet. Hierbei wird die DNF, die Trainingsdaten, die Lernrate, die Anzahl der Epochen, sowie der Modus übergeben. Der Modus dient hierbei nur zum Verändern der Gewichte, um das Verhalten des Netzes zu analysieren und ist somit kein elementarer Bestandteil der Funktionalität. Zuerst werden die Schwellenwerte und Gewichte zum initialisieren des Netzes berechnet. Danach durchläuft das Programm eine Schleife, die bei jeder Iteration eine Epoche berechnet. Nach Berechnung der Schichten wird das Ergebnis auf der Konsole ausgegeben. Schlussendlich werden die Schwellenwerte und Gewichte mithilfe der Deltas angepasst und die nächste Epoche startet. Ist die Anzahl der Epochen erreicht, stoppt das Programm das Programm und die Ergebnisse können ausgewertet werden.

Beispieldurchlauf

DNF:

```
dnf_training = np.array([
        [-1, -1, -1,  1, -1,  1, -1,  1,  0, -1],
        [-1,  0, -1,  1,  0, -1,  1,  1, -1,  0],
        [ 1, -1,  1,  0, -1, -1, -1,  1,  0,  0],
        [ 0, -1,  1,  0, -1, -1, -1,  1, -1, -1],
        [-1, -1,  0,  1, -1,  1,  1,  0,  0, -1]
  ]).astype(np.float64)
```

Als Aussageform:

$$\begin{aligned} & (\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4 \wedge \neg x_5 \wedge x_6 \wedge \neg x_7 \wedge x_8 \wedge \neg x_{10}) \\ \vee\ & (\neg x_1 \wedge \neg x_3 \wedge x_4 \wedge \neg x_6 \wedge x_7 \wedge x_8 \wedge \neg x_9) \\ \vee\ & (x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_5 \wedge \neg x_6 \wedge \neg x_7 \wedge x_8) \\ \vee\ & (\neg x_2 \wedge x_3 \wedge \neg x_5 \wedge \neg x_6 \wedge \neg x_7 \wedge x_8 \wedge \neg x_9 \wedge \neg x_{10}) \\ \vee\ & (\neg x_1 \wedge \neg x_2 \wedge x_4 \wedge \neg x_5 \wedge x_6 \wedge x_7 \wedge \neg x_{10}) \end{aligned}$$

Trainingsdaten:

```
inputs_with_p = [
    (np.array([-1, -1, -1,  1, -1,  1, -1,  1,  1, -1]), 1),
    (np.array([-1,  1, -1,  1,  1, -1,  1,  1, -1,  1]), 1),
    (np.array([ 1, -1,  1,  1, -1, -1, -1,  1,  1,  1]), 1),
    (np.array([ 1, -1,  1,  1, -1, -1, -1,  1, -1, -1]), 1),
    (np.array([-1, -1,  1,  1, -1,  1,  1,  1,  1, -1]), 1),
    (np.array([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]), -1),
    (np.array([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1]), -1),
    (np.array([ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0]), -1),
    (np.array([ 1, -1,  1, -1,  1, -1,  1, -1,  1, -1]), -1),
    (np.array([-1,  1, -1,  1, -1,  1, -1,  1, -1,  1]), -1)
]
```

Hierbei stellt das jedes np.array() Eingabevariablen dar. Der Integer ist der erwartete Output.

Die Berechnung für das erste Set an Eingabevariablen ohne Veränderung und Backpropagation wären Beispielsweise:

```
Input: [-1 -1 -1  1 -1  1 -1  1  1 -1]
Tresholds: [8.5 6.5 6.5 7.5 6.5]
Z: [ 1 -1 -1 -1 -1]
W:  [1. 1. 1. 1. 1.]
V:  -4
y: 1
```

Lassen wir nun das Programm mit leichter Veränderung der Gewichte über mehre Epochen laufen kann man erkennen wie sich die Gewichte anpassen, sobald Fehler passieren.

```
Epoche: 004 | Fehler:   2 | y: -1 | p:  1
Gewichte w: [[-2.000000 -1.100000 -1.400000 0.500000 -0.500000 0.400000 -0.100000
  0.500000 -0.200000 -0.600000]
 [-1.900000 -0.800000 -0.300000 1.300000 0.900000 -1.500000 0.100000
  0.100000 -1.500000 0.800000]
 [2.000000 -1.400000 0.300000 0.000000 -1.200000 -0.400000 -1.800000
  1.600000 0.600000 -0.700000]
 [0.700000 -0.500000 0.800000 -0.300000 -1.500000 -1.000000 -0.700000
  0.500000 -0.700000 -1.200000]
 [-0.800000 -1.600000 -1.000000 0.200000 -2.000000 1.000000 1.900000
  -0.300000 0.600000 -1.000000]]
Epoche: 005 | Fehler:  -2 | y:  1 | p: -1
Gewichte w: [[-2.200000 -1.300000 -1.600000 0.700000 -0.700000 0.600000 -0.300000
  0.700000 0.000000 -0.800000]
 [-2.100000 -1.000000 -0.500000 1.500000 0.700000 -1.300000 -0.100000
  0.300000 -1.300000 0.600000]
 [1.800000 -1.600000 0.100000 0.200000 -1.400000 -0.200000 -2.000000
  1.800000 0.800000 -0.900000]
 [0.500000 -0.700000 0.600000 -0.100000 -1.700000 -0.800000 -0.900000
  0.700000 -0.500000 -1.400000]
 [-1.000000 -1.800000 -1.200000 0.400000 -2.200000 1.200000 1.700000
  -0.100000 0.800000 -1.200000]]
```

Anfangs kommt es noch zu mehreren Fehlern, später werden es immer weniger.

```
Epoche: 001 | Fehler:   0 | y: -1 | p: -1
Epoche: 002 | Fehler:   0 | y:  1 | p:  1
Epoche: 003 | Fehler:   0 | y:  1 | p:  1
Epoche: 004 | Fehler:   0 | y: -1 | p: -1
Epoche: 005 | Fehler:   0 | y: -1 | p: -1
Epoche: 006 | Fehler:   0 | y:  1 | p:  1
Epoche: 007 | Fehler:   2 | y: -1 | p:  1
Epoche: 008 | Fehler:  -2 | y:  1 | p: -1
Epoche: 009 | Fehler:   2 | y: -1 | p:  1
Epoche: 010 | Fehler:   0 | y:  1 | p:  1
Epoche: 011 | Fehler:  -2 | y:  1 | p: -1
Epoche: 012 | Fehler:   2 | y: -1 | p:  1
Epoche: 013 | Fehler:  -2 | y:  1 | p: -1
Epoche: 014 | Fehler:   2 | y: -1 | p:  1
Epoche: 015 | Fehler:   0 | y:  1 | p:  1
Epoche: 016 | Fehler:   0 | y:  1 | p:  1
Epoche: 017 | Fehler:  -2 | y:  1 | p: -1
Epoche: 018 | Fehler:   2 | y: -1 | p:  1
Epoche: 019 | Fehler:   0 | y:  1 | p:  1
Epoche: 020 | Fehler:  -2 | y:  1 | p: -1
Epoche: 021 | Fehler:   0 | y: -1 | p: -1
Epoche: 022 | Fehler:   0 | y: -1 | p: -1
Epoche: 023 | Fehler:   0 | y: -1 | p: -1
Epoche: 024 | Fehler:   0 | y: -1 | p: -1
Epoche: 025 | Fehler:   0 | y:  1 | p:  1
```

```
Epoche: 175 | Fehler:   0 | y:  1 | p:  1
Epoche: 176 | Fehler:   0 | y:  1 | p:  1
Epoche: 177 | Fehler:   0 | y: -1 | p: -1
Epoche: 178 | Fehler:   0 | y:  1 | p:  1
Epoche: 179 | Fehler:   0 | y: -1 | p: -1
Epoche: 180 | Fehler:   0 | y: -1 | p: -1
Epoche: 181 | Fehler:   0 | y: -1 | p: -1
Epoche: 182 | Fehler:   0 | y:  1 | p:  1
Epoche: 183 | Fehler:   0 | y: -1 | p: -1
Epoche: 184 | Fehler:   0 | y: -1 | p: -1
Epoche: 185 | Fehler:   0 | y: -1 | p: -1
Epoche: 186 | Fehler:   0 | y:  1 | p:  1
Epoche: 187 | Fehler:   0 | y: -1 | p: -1
Epoche: 188 | Fehler:   0 | y:  1 | p:  1
Epoche: 189 | Fehler:   0 | y:  1 | p:  1
Epoche: 190 | Fehler:   0 | y: -1 | p: -1
Epoche: 191 | Fehler:   0 | y:  1 | p:  1
Epoche: 192 | Fehler:   0 | y:  1 | p:  1
Epoche: 193 | Fehler:   0 | y:  1 | p:  1
Epoche: 194 | Fehler:   0 | y:  1 | p:  1
Epoche: 195 | Fehler:   0 | y:  1 | p:  1
Epoche: 196 | Fehler:   0 | y:  1 | p:  1
Epoche: 197 | Fehler:   0 | y:  1 | p:  1
Epoche: 198 | Fehler:   0 | y:  1 | p:  1
Epoche: 199 | Fehler:   0 | y: -1 | p: -1
Epoche: 200 | Fehler:   0 | y:  1 | p:  1
```

Auswertung

Aufgabe c):

Wenn die Gewichte und Schwellenwerte initial so gesetzt werden, wie in der Aufgabenstellung beschrieben, ist die realisierung der DNF mit den gegebenen Variablen schon bei dem ersten Durchlauf des Netzes richtig. Das Ergebnis ist immer korrekt, was bedeutet, dass sich die Gewichte und Schwellwerte nicht mehr anpassen.

Aufgabe d) Mit leichter Veränderung der Schwellenwerte und Gewichte:

Mit leicht veränderten Werten macht das Netz von Anfang an wenig Fehler. Bei mehrfacher Durchführung des Programms verhält sich das Netz immer ähnlich.

Aufgabe d): Mit zufälligen Schwellenwerten und Gewichten:

Das Netz lernt tatsächlich mit der Zeit die Funktion richtig zu realisieren. Die meisten Fehler passieren in den ersten 25 Epochen vor, danach eher selten. Nach 50 - 100 Epochen kommen so gut wie keine Fehler vor. Allgemein kann man sagen, dass umso mehr Epochen das Netz durchgelaufen hat, umso weniger Fehler passieren.