Dokumentation Programmentwurf 1

Yannik Angeli, Nils Kubousek, Lukas Richter, Diego Rubio Carrera

Datum: 2024-04-25

Diese Dokumentation ist auch als PDF verfügbar: Dokumentation.pdf

Inhaltsverzeichnis:

Projektübersicht

Dieses Projekt implementiert ein zweischichtiges, vorwärtsgetriebenes neuronales Netzwerk, das disjunktive Normalform (DNF)-Formeln der Aussagenlogik realisieren kann. Die Implementierung umfasst eine Netzwerkarchitektur, die logische Operationen auf Gewichte und Schwellenwerte des neuronalen Netzes abbildet, sowie einen Backpropagation-Algorithmus für das Training.

Architektur

Das neuronale Netzwerk besteht aus drei Schichten:

- **Eingabeschicht**: Nimmt binäre Eingaben entgegen, die Wahrheitswerte repräsentieren (-1 für falsch, 1 für wahr)
- **Zwischenschicht**: Enthält Neuronen, die Monome (Konjunktionen von Literalen) darstellen
- **Ausgabeschicht**: Ein einzelnes Neuron, das die gesamte DNF-Formel (Disjunktion von Monomen) repräsentiert

Codestruktur

Das Projekt ist in mehrere Module organisiert, welche sich im Verzeichnis `src` befinden:

net.py

Enthält die zentrale `DNFNet`-Klasse, die das neuronale Netzwerk implementiert. Zu den Hauptfunktionen gehören:

1. `__init__(self, input_length: int = 10, num_monomers: int = 5, learning_rate: float = 0.1)`: Initialisierung mit konfigurierbarer Eingabegröße, Anzahl der Monome und Lernrate
2. `inference(self, inputs: list[int]) -> tuple[int, list[int]]`: Vorwärtspropagation zur Berechnung der Netzwerkausgabe
3. `backpropagation(self, inputs: list[int], target: int, result: int, monomer_activations: list[int]) -> None`: Backpropagation-Algorithmus zur Aktualisierung von Gewichten und Schwellenwerten anhand des Eingabemusters, der Zielausgabe und der tatsächlichen Ergebnisse des Netzwerks.
4. `train(self, inputs: list[int], target: int) -> tuple[int, list[int]]`: Trainiert das Netzwerk mit einem gegebenen Eingabemuster und der entsprechenden Zielausgabe, kombiniert also die Methoden `inference` und `backpropagation`
5. `activation(self, num: float | int)`: Implementierung der Signumfunktion als Aktivierungsfunktion
6. `__str__(self) -> str`: Dient der Formatierung der Netzwerkkonfiguration für die Ausgabe
7. `__call__(self, inputs: list[int], target: int = 0, train: bool = False) -> tuple[int, list[int]]`: Ermöglicht die Aufruf des Netzwerks wie eine Funktion, um Vorhersagen zu treffen. Erlaubt dabei neben auch die Angabe eines Booleans `train`, der angibt, ob das Netzwerk im Trainings- oder Inferenz-Modus arbeiten soll
8. `__eq__(self, value: object) -> bool`: Dient der Vergleichbarkeit von Netzwerken anhand deren Gewichten und Schwellwerten, gibt False zurück, wenn `value` kein `DNFNet` ist
9. `shape(self) -> tuple[int, int]`: Property, die die Netzwerksdimensionen (Eingabelänge und Anzahl der Monome) als Tupel zurückgibt

train.py

Implementiert die überwachte Trainingsfunktionalität durch die `supervised_train`-Funktion, die:

- Ein korrektes Netzwerk (Referenz)
- und ein Trainingsnetzwerk entgegennimmt
- sowie die Spezifizierung eines Epochenlimits erlaubt

- Das Netzwerk iterativ mit allen möglichen Eingabekombinationen trainiert, wobei die korrekte Ausgabe durch das übergebene `correct_net` berechnet wird
- Die Anzahl der falschen Netzwerkvorhersagen pro Epoche verfolgt
- Fehler auf Monomerebene für detaillierte Analysen überwacht
  - Dabei ist zu beachten, dass die Reihenfolge der Monome im Trainingsnetzwerk nicht zwingend mit der des Referenznetzwerks übereinstimmen muss, die hier erfassten Werte also begrenzte Aussagekraft haben
- Konvergenzkriterien und Maximallimits für Epochen implementiert - erfolgt eine gesamte Epoche ohne Aktualisierung der Gewichte, wird die Konvergenz als erreicht angesehen und das Netzwerk nicht weiter trainiert. Anderenfalls wird es bis zum Erreichen des Epochenlimits trainiert.

`helpers.py`

Bietet Hilfsfunktionen:

1. `random_adjustment(factor: float = 1) -> float`: Gibt einen zufälligen Wert zwischen -1 und 1 oder -`factor` und `factor` zurück
2. `plot_network_results(network_misses: list[int|float]) -> None`: Erstellt ein Diagramm der Netzwerkfehler pro Epoche
3. `plot_monomer_results(monomer_misses_per_epoch: list[list[int|float]]) -> None`: Erstellt ein Diagramm der Monomerfehler pro Epoche - wie in `supervised_train` erwähnt, mit begrenzter Aussagekraft
4. `sma(series: list[int], window: int) -> list[float]`: Berechnet den gleitenden Durchschnitt (SMA) über eine bestimmte Fenstergröße in einer Zeitreihe
5. `plot_combined_sma(incorrect_per_epoch: list[int], monomer_misses_per_epoch: list[list[int]], window: int) -> None`: Erstellt ein Diagramm des gleitenden Durchschnitts der Netzwerk- und Monomerfehler pro Epoche
6. `visualize_results(network_misses: list[int], monomer_misses_per_epoch: list[list[int]], window: int = 100) -> None`: Visualisiert die Ergebnisse des Trainings mit Hilfe der obigen drei Funktionen
7. `plot_several_networks(network_misses: list[list[int]], names: list[str], window: int = 1) -> None`: Erstellt ein Diagramm vom SMA der Netzwerkfehler pro Epoche für mehrere Netzwerke oder für dasselbe Netzwerk mit verschiedenen Lernraten

Netzwerkbetrieb

Initialisierung

Das Netzwerk wird grundsätzlich mit zufälligen Parametern initialisiert. Spezifische Gewichte müssen manuell per Zugriff auf die Netzwerkattribute gesetzt werden.

Inferenz

Während der Vorwärtspropagation:

1. Eingangssignale werden an jedem Monomerneuron gewichtet und summiert
   - Die Vorzeichenfunktion wird angewendet
2. Monomerausgaben werden am Ausgabeneuron gewichtet und kombiniert
   - Die Vorzeichenfunktion wird angewendet
3. Die endgültige Ausgabe repräsentiert den Wahrheitswert der Formel, die durch das Netzwerk dargestellt wird

Mathematisch ausgedrückt gilt also:

- n Aussagenvariablen $x_1, \ldots, x_n$ (Eingangsneuronen)
- Werte -1 (falsch) und 1 (wahr)
- m Monome $z_1, \ldots, z_m$ (Neuronen in Zwischenschicht)
- Ein Ausgabeneuron y
- Aktivierungsfunktion $sgn(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$
- Propagation in Vektorform:
  - $\vec{z} = sgn(w \cdot \vec{x} - \vec{v})$
  - $y = sgn(W \cdot \vec{z} - V)$
  - mit Gewichtsmatrizen:
    - $w \in \mathbb{R}^{m \times n}$
    - $W \in \mathbb{R}^{1 \times m}$
  - und Schwellwerten:
    - $\vec{v} \in \mathbb{R}^m$
    - $V \in \mathbb{R}$

Training

Der Backpropagation-Algorithmus aktualisiert die Gewichte folgendermaßen:

- Lernalgorithmus mit Zielmuster p und Lernrate η:
- $\Delta W_{1j} = \mu \cdot (p - y) \cdot z_j$
- $\Delta w_{jk} = \mu \cdot W_{1j} \cdot (p - y) \cdot x_k$
- $\Delta V = -\mu \cdot (p - y)$
- $\Delta v_j = -\mu \cdot (p - y) \cdot W_{1j}$

Verwendung

Das Netzwerk in `net.py` und die Module `helpers.py` sowie `train.py` können verwendet werden, um:

1. Eine DNF-Formel direkt mit berechneten Gewichten zu implementieren
2. Eine DNF-Formel durch überwachtes Training zu lernen, wobei jedoch bereits Kenntnis über die korrekten Parameter vorausgesetzt wird, sodass das Referenzmodell erstellt werden kann
3. Die Lernleistung mit verschiedenen Parametern zu vergleichen
4. Den Lernprozess und die Netzwerkkonvergenz zu visualisieren

Beispiel

```
from src.net import DNFNet
from src.train import supervised_train
from src.helpers import visualize_results

# Referenznetzwerk mit vorberechneten Gewichten erstellen
correct_net = DNFNet(input_length=10, num_monomers=5)


# Parameter entsprechend der DNF-Formel setzen:
correct_model.monomer_weights = [
    [1, 1, 1, -1, 0, 0, 0, 0, 0, 0],
    [0, 1, 1, 1, -1, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, -1, 0, 0, 0, 0],
    [0, 0, 0, 1, 1, 1, -1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, -1, -1, -1]
]

correct_model.monomer_biases = [4, 4, 4, 4, 4]

correct_model.output_weights = [1, 1, 1, 1, 1]
```

```
correct_model.output_bias = -3



# Zu trainierendes Netzwerk erstellen
train_net = DNFNet(input_length=10, num_monomers=5, learning_rate=0.1)

# Netzwerk trainieren
incorrect_per_epoch, monomer_misses = supervised_train(correct_net, train_net)

# Ergebnisse visualisieren
visualize_results(incorrect_per_epoch, monomer_misses)
```