Brandenburgische
Technische Universität
Cottbus - Senftenberg

Faculty of Mathematics, Computer Science, Physics, Electrical Engineering and Information Technology

Institute of Computer Science

COMPUTER SCIENCE REPORTS

Report 01/24

January 2024

ANN-Partitionen und CQQL-Ausdrücke

Ingo Schmitt

Computer Science Reports
Brandenburg University of Technology Cottbus - Senftenberg
ISSN: 1437-7969
https://doi.org/10.26127/BTUOpen-6606
Send requests to: BTU Cottbus - Senftenberg
Institut für Informatik
Postfach 10 13 44
D-03013 Cottbus

ANN-Partitionen und CQQL-Ausdrücke

Ingo Schmitt[0000−0002−4375−8677]

Brandenburgische Technische Universität Cottbus-Senftenberg, Cottbus, Germany
schmitt@b-tu.de
https://www.b-tu.de/fg-dbis

Zusammenfassung. Gegeben sei für ein binäres Klassifikationsproblem ein künstliches, neuronales Netzwerk `ann` bestehend aus ReLU-Knoten und linearen Schichten (convolution, pooling, fully connected). Das Netzwerk `ann` sei mit hinreichender Genauigkeit an Hand von Trainingsdaten trainiert. Wir werden zeigen, dass ein solches Netzwerk in verschiedene Partitionen des Eingaberaums zerlegt werden kann, wobei jede Partition eine lineare Abbildung der Eingabewerte auf einen klassifizierenden Ausgabewert repräsentiert. Im Weiteren gehen wir von einem einfachen Netzwerk `ann` aus, bei dem die Eingangswerte Mintermen von Attributwerten entsprechen. Einfach ist ein Netzwerk, wenn es für eine geringe Anzahl von Attributen trainiert wurde und die Anzahl der ReLU-Knoten ebenfalls gering ist. In der Arbeit wird gezeigt, dass jede lineare Partition durch einen CQQL-Ausdruck beschrieben werden kann. Ein CQQL-Ausdruck lässt sich mit Hilfe von Quantenlogik-inspirierten Entscheidungsbäumen beschreiben.

Schlüsselwörter: Quantum logic · Artificial neural networks

1 Ausgangslage

Für ein binäres Klassifikationsproblem sei eine Menge $O = \{(x_i, y_i)\}$ von Eingabe-Ausgabe-Paaren gegeben. Eine Teilmenge $TR \subseteq O$ stellt die Trainingsmenge dar, wobei $x_i \in \mathbb{R}^n$ und $y_i \in \{0, 1\}$ gelten. Zur Lösung des binären Klassifikationsproblems gehen wir von einem künstlichen, neuronalen Netzwerk `ann` aus, welches aus ReLU-Knoten und linearen Schichten (pooling, convolution, fully connected) konstruiert wurde. Das Netzwerk `ann` habe genau ein Ausgabeneuron. Im Folgenden gehen wir von einer kleinen Anzahl n von Eingabeattributen sowie von einer kleinen Anzahl von ReLU-Knoten aus.

Im Folgenden wollen wir die Attributwerte $x_i[j]$ als logische CQQL-Werte aus dem Intervall $[0, 1] \subseteq \mathbb{R}$, siehe [1,2,5], interpretieren. Dazu verwenden wir monoton ansteigende Funktionen $m_j()$ mit

$$m_j(x_i[j]) \in [0, 1]$$

für $j = 1, \ldots, n$. Weiterhin wollen wir das Netzwerk `ann` als logische Ausdrücke interpretieren und gehen daher davon aus, dass die Eingabewerte des Netzwerks

Mintermen $mt_i[k]$ an Stelle von Attributwerten $m_j(x_i[j])$ entsprechen. Minterme sind entsprechend [4,3] wie folgt definiert:

$$mt_i[k] := \Pi_{j=1}^{n}(1 - m_j(x_i[j])^{1-b_j} m_j(x_i[j])^{b_j} \in [0,1],$$

wobei $k = 0, \ldots, 2^n - 1$ gilt und $b = b_1 \ldots b_n$ der jeweilige Bit-Code von k ist. Minterm-basierte Trainingsdaten bezeichnen wir mit $TR_{mt} := \{(mt_i, y_i)\}$. Ein künstliches, neuronales Netzwerk $\texttt{ann}$ auf der Grundlage von Mintermen zur Lösung eines binären Klassifikationsproblems hat 2^n Eingabeneuronen und 1 Ausgabeneuron. Wir gehen davon aus, dass das Netzwerk $\texttt{ann}$ mittels TR_{mt} und einer Verlustfunktion, etwa MSE, mit hoher Genauigkeit trainiert wurde. Zur Vorhersage von $y_i \in \{0,1\}$ an Hand von $\texttt{ann}(mt_i)$ verwenden wir eine Schwellenwertfunktion $th_\tau(\texttt{ann}(mt_i))$ mit dem Ausgabeschwellenwert τ, wobei

$$th_\tau(x) := \begin{cases} 1 \text{ if } x > \tau \\ 0 \text{ sonst} \end{cases}$$

gilt. Wie wird ein geeigneter Ausgabeschwellenwert τ gewählt? Dazu ermitteln wir den Minimalwert der 1-Objekte

$$\min_1 = \min_{(mt_i,1) \in TR_{mt}} ann(mt_i)$$

sowie den Maximalwert der 0-Objekte

$$\max_0 = \max_{(mt_i,0) \in TR_{mt}} ann(mt_i).$$

Im Fall von $\max_0 < \min_1$ sind die 0-Objekte perfekt von den 1-Objekten separiert und wir setzen $\tau := (\max_0 + \min_1)/2$. Ansonsten wählen wir einen Wert aus dem Intervall $[\min_1, \max_0]$ aus. Dazu verwenden wir den Ausgabewert des Trainingsobjekts, welches die höchste Genauigkeit erzeugt:

$$\tau := \underset{\substack{\tau_{mt_i} := \texttt{ann}(mt_i) \\ (mt_i, \_) \in TR_{mt} \\ \tau_{mt_i} \in [\min_1, \max_0]}}{\arg\max} acc(\texttt{ann}, \tau_{mt_i}, TR_{mt}) \tag{1}$$

wobei

$$acc(\texttt{ann}, \tau_{mt_i}, TR_{mt}) = |\{(mt_j, y_j) \in TR_{mt} | y_j = th_{\tau_{mt_i}}(\texttt{ann}(mt_j))\}|/|TR_{mt}|$$

gilt.

Mit Hilfe der Testdaten $TE_{mt} = O \setminus TR_{mt}$ kann nun die finale Genauigkeit des trainierten Netzwerks $\texttt{ann}$ bestimmt werden. Im Folgenden gehen wir von einem trainierten Netzwerk $\texttt{ann}$ auf Mintermen mit hoher Genauigkeit auf den Testdaten aus.

2 ANN-Partitionen

Angenommen, das Netzwerk $\texttt{ann}$ hat l ReLU-Knoten. Dann lässt sich für jedes Trainingsobjekt mt_i der Status der ReLU-Knoten mittels $relu_i$ zuweisen. Dabei

ist $relu_i[m] \in \{0,1\}$ der m-te ReLU-Knotenstatus mit $m = 1,\ldots,l$, welcher auf Eins (aktiver Knoten) gesetzt ist, wenn sein Eingabewert eine nicht-negative Zahl ist. Ansonsten ist der Status auf Null (inaktiv) gesetzt. Wir interpretieren $relu_i$ als Bit-Code für den ganzzahligen Wert $p_i \in \{0,\ldots,2^l-1\}$:

$$p_i := relu_i[1]\ldots relu_i[l].$$

Den Wert p_i eines i-ten Trainingsobjekts bezeichnen wir als *Partitionszahl*. Basierend auf äquivalenten Partitionszahlen zerlegen wir die Trainingsdaten in Äquivalenzklassen, die wir als *Trainingspartitionen* bezeichnen:

$$TR_{mt} = \bigcup_{p=0}^{2^l-1} \{(mt_i, y_i) | p_i = p\}.$$

Alle Trainingsobjekte der selben Trainingspartition besitzen also die gleiche Menge von aktiven und inaktiven ReLU-Knoten.

Analog zur Partitionierung der Trainingsdaten kann die Menge der hypothetischen Eingabeobjekte in Partitionen zerlegt werden. Wir konzentrieren uns im Folgenden auf eine bestimmte Partitionszahl p. Wir reduzieren das Netzwerk `ann` mittels der Partitionszahl p zu einer Netzwerkpartition $\texttt{ann}_p$, indem aktive ReLU-Knoten durch das konstantes Gewicht Eins ersetzt werden. Inaktive ReLU-Knoten werden durch Null ersetzt. Damit besteht jede Netzwerkpartition $\texttt{ann}_p$ ausschließlich aus linearen Schichten. Da die Hintereinanderausführung von linearen Operationen selbst wieder eine lineare Operation ist, können wir $\texttt{ann}_p$ mittels einer Linearkombination auf den Minterm-Werten des Eingabeobjekts i mit $p_i = p$ beschreiben:

$$\texttt{ann}(mt_i) = \texttt{ann}_p(mt_i) = \sum_{k=0}^{2^n-1} mw_p[k] * mt_i[k]. \tag{2}$$

Der Vektor $mw_p \in \mathbb{R}^{2^n}$ enthält die Minterm-Gewichte der Linearkombination für Objekte der Partitionszahl p. Damit repräsentiert mw_p die Netzwerkpartition $\texttt{ann}_p$. Dies bedeutet, dass eine Netzwerk `ann`, bestehend aus l ReLU-Knoten sowie linearen Schichten (convolution, pooling, fully connected) funktional mittels 2^l vielen Linearkombinationen mw_p von Minterm-Werten beschrieben werden kann.

3 CQQL-Ausdruck einer Netzwerkpartition

Ein Quantumlogik-inspirierter, binärer CQQL-Klassifikator, siehe [3], ist definiert durch

$$cl_e^\tau(o_i) := th_\tau([e]^{o_i}),$$

wobei e ein logischer CQQL-Ausdruck auf n Eingabeattributen ist und $[e]^{o_i} \in [0,1]$ die arithmetische Auswertung des logischen Ausdrucks auf den Attributwerten des Objekts o_i bezeichnet. In CQQL gelten die Booleschen Gesetze. Jeder

logische Ausdruck e kann in der vollständigen, disjunktiven Normalform ausgedrückt werden. Deswegen lässt sich jeder logischer Ausdruck e durch eine Menge von Mintermen (die aktiven Minterme) als Teilmenge aller 2^n Mintermen identifizieren. Wenn wir aktive Minterme mit 1 und inaktive Minterme mit 0 verknüpfen, wird ein logischer Ausdruck e durch einen Bitvektor $mw^e \in \{0,1\}^{2^n}$ beschrieben. Zusammen mit den Mintermwerten mt_i eines Eingabeobjekts o_i ist die CQQL-Auswertung des logischen Ausdrucks e durch:

$$[e]^{o_i} = \sum_{k=0}^{2^n-1} mw^e[k] * mt_i[k] \tag{3}$$

definiert. Die Auswertung $[e]^{o_i}$ ist keine Boolesche Auswertung, da die Eingabewerte $mt_i[k] \in [0,1]$ nicht als diskrete, Boolesche Werte interpretierbar sind. Trotzdem gelten für CQQL-Ausdrücke im Gegensatz zur Fuzzy-Logik alle Booleschen Gesetze.

Der Bitvektor $mw^e \in \{0,1\}^{2^n}$ eines logischen Ausdrucks e kann gedanklich mit einem n-dimensionalen Hyperwürfel mit 2^n Hyperecken verbunden werden, wobei jede Hyperecke (Minterm) entweder aktiv oder inaktiv ist. Ein logischer Ausdruck e für ein binäres Klassifikationsproblem drückt damit aus, in welcher Hyperecke des Hyperwürfels überwiegend 1-Objekte räumlich zu finden sind.

Die CQQL-Auswertung nach Formel 3 ist sehr ähnlich zu der Auswertung einer Netzwerkpartition nach Formel 2. Der Unterschied liegt darin, dass die mw_p-Werte reelle Zahlen und die mw^e-Werte Bitwerte sind. Um eine Netzwerkpartition mw_p durch einen logischen Ausdruck (oder durch mehrere Ausdrücke) mw^e interpretieren zu können, benötigen wir eine geeignete Abbildung, welche die Auswertung nach Formel 2 durch eine oder mehrere Auswertungen nach Formel 3 hinreichend gut approximiert.

4 Abbildung von reellwertigen Mintermgewichten auf bitwertige Gewichte

Die Formel 2 entspricht dem Skalarprodukt zwischen mt_i und mw_p. Die lineare Algebra lehrt uns, dass das Skalarprodukt linear bezüglich seiner Argumente ist. Daher skalieren wird alle Mintermgewichte $mw_p[k] \in \mathbb{R}$ mittels einer monoton ansteigenden Funktion f auf das Einheitsintervall

$$mw_p[k]' := f(mw_p[k]) \in [0,1]$$

und erhalten als normiertes Auswertungsresultat folgende Äquivalenz:

$$\mathtt{ann}'_p(mt_i) := \mathtt{ann}_p(f(mt_i)) = f(\mathtt{ann}_p(mt_i)).$$

Da außerdem $\sum_k mt_i[k] = 1$ und $mt_i[k] \geq 0$ gelten, erhalten wir

$$\mathtt{ann}'_p(mt_i) \in [0,1].$$

Die monoton ansteigende Funktion f bewahrt die Reihenfolge der Auswertungsergebnisse für alle $i_1, i_2, \tau, \tau' := f(\tau)$:

$$\texttt{ann}_p(mt_{i_1}) \leq \tau \leq \texttt{ann}_p(mt_{i_2}) \Leftrightarrow \texttt{ann}'_p(mt_{i_1}) \leq \tau' \leq \texttt{ann}'_p(mt_{i_2}).$$

Aus diesem Grund ändert die Normierung auf das Einheitsintervall nicht die Genauigkeit des Klassifikators. Im Folgenden gehen wir von normierten Mintermgewichten aus, ohne dies extra durch Apostrophe anzuzeigen.

Für die Konstruktion von mw_p^e eines logischen Ausdrucks e für eine Partitionszahl p muss für jeden Minterm k entschieden werden, ob $mw_p^e[k]$ aktiv (1) oder inaktiv (0) ist. Wir gehen davon aus, dass ein hoher Wert $mw_p[k]$ zu einem hohen Wert mw_p^e korrespondiert. Dies entspricht folgender Monotonie-Eigenschaft für alle Minterme $k_1, k_2 \in \{0, \ldots, 2^n - 1\}$:

$$mw_p[k_1] \geq mw_p[k_2] \Leftrightarrow mw_p^e[k_1] \geq mw_p^e[k_2].$$

Als Konsequenz sortieren wir alle Werte $mw_p[k]$ absteigend mittels einer Permutation ()

$$mw_p[(0)] \geq mw_p[(1)] \geq \ldots \geq mw_p[(2^n - 1)]$$

und suchen eine geeignete Rangposition $\tau_{mt} \in \{0, \ldots, 2^n - 1\}$ für die Entscheidung über aktiv oder inaktiv: $mw_p^e[(k)] := 1$ wenn $k \leq \tau_{mt}$, ansonsten $mw_p^e[(k)] := 0$. Eine Rangposition τ_{mt} ist geeignet, wenn sie und der dazu ermittelte Schwellenwert τ die beste Genauigkeit des resultierenden Klassifikators cl_e^τ auf den Trainingsdaten liefert. Für die Ermittlung des Schwellenwerts τ nutzen wir Formel 1. Zum Finden der geeigneten Rangposition laufen wir durch alle Positionen, beginnend mit der Position 0, aktivieren die entsprechenden Mintermgewichte und berechnen die positionsabhängige Genauigkeit. Zuvor setzen wir alle Mintermgewichte auf inaktiv (0). Abbildung 1 zeigt den entsprechenden Algorithmus. Die Schrittweite `step` ist dabei auf 1 gesetzt.

Als Verfeinerung werden nach dem ersten Durchlauf die aktualisierten Mintermgewichte aufsteigend sortiert und die Minterme erneut durchlaufen, um Minterme schrittweise zu deaktivieren, sofern dies eine bessere Genauigkeit liefert. Abbildung 2 zeigt den entsprechenden Algorithmus (Aufruf mit $level := 0$ und $mw_p^e := 0$).

5 Optimale Mintermgewichte für unterschiedliche Stufen

Offensichtlich kann ein Bitvektor mw_p^e einen reellwertigen Vektor mw_p im Allgemeinen nur unzureichend nachbilden. Unsere Idee besteht darin, sich mit mehreren Bitvektoren für unterschiedliche Stufen dem reellwertigen Vektor schrittweise anzunähern. Im ersten Schritt (Stufe 0) verwendeten wir eine Schrittweite von Eins: eine Eins wird entweder auf Gewichte addiert oder abgezogen, sofern das Ergebnis im Einheitsintervall liegt und eine Verbesserung der Genauigkeit liefert. Im zweiten Schritt (Stufe 1) halbieren wird die Schrittweite, sei beträgt also 1/2. Der zweite Schritt baut auf dem ersten Schritt auf, daher sortieren wir die Differenzen zwischen $mw_p[k]$ und $mw_p^e[k]$ vor dem nächsten Durchlauf. In

```
function optimize_weights_for_perm(mw_p^e, (), step)
  best_acc := 0
  for k := 0 to 2^n − 1
    if mw_p^e[(k)] + step ∈ [0,1]
      mw_p^e[(k)] := mw_p^e[(k)] + step
    find best τ for mw_p^e and
      let acc be its generated accuracy
    if acc > best_acc
      best_acc := acc
      best_mw_p^e := mw_p^e
  return best_mw_p^e
```

Abb. 1. Optimale Mintermgewichte basierend auf einer Permutation ()

```
function optimize_weights_for_level(mw_p, mw_p^e, level)
  step := 2^level
  create permutation () by ordering (mw_p[k] − mw_p^e[k]) descendingly
  mw_p^e := optimize_weights_for_perm(mw_p^e, (), step)
  create permutation () by ordering (mw_p[k] − mw_p^e[k]) ascendingly
  mw_p^e := optimize_weights_for_perm(mw_p^e, (), −step)
  return mw_p^e
```

Abb. 2. Optimale Mintermgewichte für eine vorgegebene Stufe

einem dritten Schritt (Stufe 2) wird erneut die Schrittweite halbiert. Wir iterieren solange, bis wir eine vorgegebene Maximalstufe (`max_level`) erreicht haben. Abbildung 3 zeigt den entsprechenden Algorithmus.

```
function optimize_weights(mw_p, max_level)
  for all k = 0,...,2^n
    mw_p^e[k] := 0
  for level := 0 to max_level
    mw_p^e := optimize_weights_for_level(mw_p, mw_p^e, level)
  return mw_p^e
```

Abb. 3. Optimale Mintermgewichte

6 Verschwindender Ausgabefehler

Für einen gegebenen Wert max_level und einen Minterm k können wir die obere Grenze des absoluten Fehlers zwischen $mw_p[k]$ und $mw_p^e[k]$ mit $2^{-(max\_level+1)}$ abschätzen. Darauf aufbauend, erhalten wir die Obergrenze für den absoluten

Fehler auf dem Ausgabewert eines Objekts i, welcher sich mittels des Skalarprodukts über den Mintermen berechnet. Wegen $\sum_k mt_i[k] = 1$ erhalten wir die selbe Obergrenze von $2^{-(max\_level+1)}$ für den absoluten Fehler zwischen $[e]^{o_i}$ und $\texttt{ann}_p(mt_i)$ mit $p_i = p$. Der Ausgabefehler verschwindet also exponentiell mit steigendem Wert für max_level.

7 Interpretation des logischen CQQL-Ausdrucks e

Nach der Ermittlung eines logischen CQQL-Ausdrucks e in Form von mw_p^e für eine Partition p möchte man die Möglichkeiten der Logik nutzen, um e zu interpretieren. In Abhängigkeit von max_level enthält mw_p^e Gewichtswerte auf verschiedenen diskreten Stufen:

$$l = 0/2^{max\_level}, 1/2^{max\_level}, \ldots, 2^{max\_level}/2^{max\_level}.$$

Aufbauend auf den Stufen lassen sich nun die Mintermgewichte $mw_{p,l}^e$ pro Stufe l als Menge von jeweils aktiven Mintermen festlegen:

$$mw_{p,l}^e := \{k | mw_p^e[k] = l\}$$

Mittels dieser gestuften Mintermgewichte $mw_{p,l}^e$ berechnen wir $[e]^{o_i}$ für ein Objekt o_i der Partition $p_i = p$ wie folgt:

$$[e]^{o_i} = \sum_k mw_p^e[k] * mt_i[k] = \sum_l l * \sum_{k \in mw_{p,l}^e} mt_i[k].$$

Wir erhalten also eine Menge aktiver (1) Minterme pro Stufe zusammen mit einem Stufengewicht l. Jede Stufe kann somit als CQQL-Ausdruck mit einem Gewichtswert l verstanden werden. Alle logischen CQQL-Ausdrücke $e_{p,l}$ approximieren damit das Netzwerk $\texttt{ann}$ und können zur Logikinterpretation verwendet werden.

Ein einzelner Ausdruck $e_{p,l}$ kann für ein gutes Verständnis in Form eines Quantenlogik-inspirierten Entscheidungsbaums *qldt* entsprechend [4] visualisiert werden. Attributentscheidung nahe der Wurzel etwa zeigen implizit eine höhere Wichtigkeit als wurzelferne Attributentscheidungen an.

Eine andere Variante der Interpretation ist ein Test einer logikbasierten Hypothese e_{test} gegenüber einem Logikausdruck $e_{p,l}$. Auf der Grundlage der Mintermmengen mw_{test}^e, $mw_{test}^{\neg e}$, $mw_{p,l}^e$ sowie $mw_{p,l}^{\neg e}$ und deren Schnittmengen lassen sich Maße für die Mintermüberlappung in Analogie zu Genauigkeit, Präzision und Recall berechnen. Genauigkeit etwa lässt sich wie folgt berechnen:

$$acc(e_{p,l}, e_{test}) = \frac{|mw_{p,l}^e \cap mw_{test}^e| + |mw_{p,l}^{\neg e} \cap mw_{test}^{\neg e}|}{2^n}.$$

Weiterhin können auch Logikausdrücke über Partitionen hinweg analysiert werden. Zum Beispiel können Minterme der Stufe $l = 1$ in allen Partitionen auftauchen und sind daher essentiell. Auch Aggregationen wie die Durchschnittsberechnung sind denkbar. Da Partitionen unterschiedliche Größen bzgl. Anzahl entsprechender Trainingsobjekte aufweisen können, macht ein gewichteter Durchschnitt Sinn.

Literatur

1. Schmitt, I.: Quantum query processing: unifying database querying and information retrieval. Citeseer (2006)
2. Schmitt, I.: Qql: A db&ir query language. The VLDB journal **17**(1), 39–56 (2008)
3. Schmitt, I.: QLC: A quantum-logic-inspired classifier. In: Desai, B.C., Revesz, P.Z. (eds.) IDEAS'22: International Database Engineered Applications Symposium, Budapest, Hungary, August 22 - 24, 2022. pp. 120–127. ACM (2022). https://doi.org/10.1145/3548785.3548790, https://doi.org/10.1145/3548785.3548790
4. Schmitt, I.: QLDT: A decision tree based on quantum logic. In: Chiusano, S., Cerquitelli, T., Wrembel, R., Nørvåg, K., Catania, B., Vargas-Solar, G., Zumpano, E. (eds.) New Trends in Database and Information Systems - ADBIS 2022 Short Papers, Doctoral Consortium and Workshops: DOING, K-GALS, MADEISD, MegaData, SWODCH, Turin, Italy, September 5-8, 2022, Proceedings. Communications in Computer and Information Science, vol. 1652, pp. 299–308. Springer (2022). https://doi.org/10.1007/978-3-031-15743-1_28, https://doi.org/10.1007/978-3-031-15743-1_28
5. Schmitt, I., Baier, D.: Logic based conjoint analysis using the commuting quantum query language. In: Algorithms from and for Nature and Life, pp. 481–489. Springer (2013)