© vegefox.com | Adobestock

Inferenz in autonomen Fahrzeugen – ein deterministischer Ansatz

Sichere neuronale Netze

Bei der Entwicklung von Software für sicherheitskritische Systeme wird sichergestellt, dass die Software räumlich und zeitlich deterministisch ist. Wie funktionieren neuronale Netze hier? Wie gut eignen sich GPU und Parallelverarbeitungs-Engines für die sichere Inferenz neuronaler Netze?

Lucas Fryzek

Bei autonomen Systeme in Fahrzeugen kommen künstliche neuronale Netze zum Einsatz. Steigen Menschen in ein Fahrzeug ein, erwarten sie Sicherheit. Dafür gibt es aktive und passive Sicherheitssysteme wie Sicherheitsgurte, automatische Bremssysteme oder Airbags. Doch wie lässt sich beweisen, dass auch die autonomen Fahrsysteme, die auf neuronalen Netzen basieren, sicher sind? Wissenschaftler haben maßgeblich Methoden erforscht um nachzuweisen, dass ein gegebenes neuronales Netz sicher und zuverlässig ist, aber es wurde wenig dazu geforscht, wie das neuronale Netz sicher und deterministisch ausgeführt werden kann. Um die Vorteile neuronaler Netze in diesen sicheren Systemen nutzen zu können, bedarf es Methoden, um die neuronalen Netze auf sichere Weise einzusetzen.

Gängige neuronale Netze wie MobileNets werden mit Software-Bibliotheken wie TensorFlow und PyTorch erstellt. Diese Bibliotheken ermöglichen es, ihr Design einfach zu iterieren. Allerdings sind diese Bibliotheken nicht auf funktionale Sicherheit ausgelegt, keine von ihnen bietet Zertifizierungsartefakte für Standards wie ISO 26262. Tatsächlich verlassen sich viele dieser Bibliotheken auf Software-Plattformen wie Python, die eine Zertifizierungs-Barriere im Automobilbereich darstellen. Entwickler müssen sicherstellen, dass Software für sicherheitskritische Systeme sowohl räumlich als auch zeitlich deterministisch ist.

Wie neuronale Netze funktionieren

Bei einem einfachen neuronalen Netz lässt sich jeder Knoten durch $f_{i,j}$ darstellen, wobei i die Schicht im Netzwerk und j den j-ten Knoten in dieser Schicht repräsentiert, wie in **Bild 1** zu sehen ist.

Zur Inferenz mit einem neuronalen Netz ist ein grundlegender Vorgang auf jedem Knoten im Netzwerk anzuwen-

den. Dieser besteht darin, eine gewichtete Summe aller Eingänge zu einem Knoten im Netzwerk zu nehmen und dann eine Aktivierungsfunktion σ auf diese gewichtete Summe anzuwenden. Eine gängige Darstellung der Gleichung für ein neuronales Netz zeigt **Bild 2**.

Hier wird die Aktivierungsfunktion σ auf jede Komponente des resultierenden Vektors angewendet, wobei f_i ein Vektor ist, der den Aktivierungswert jedes Knotens in der i-ten Schicht des Netzwerks abbildet. $w_{j,n}$ stellt das Gewicht jeder Verbindung dar, die in die i-te Schicht des Netzwerks gelangt. Der Bias für jeden Knoten in der i-ten Schicht wird von b_n repräsentiert. Um dieses neuronale Netz auszuführen, muss nun f_i für jede Schicht im Netzwerk errechnet werden und die letzte Schicht des Netzwerks repräsentiert das Ergebnis.

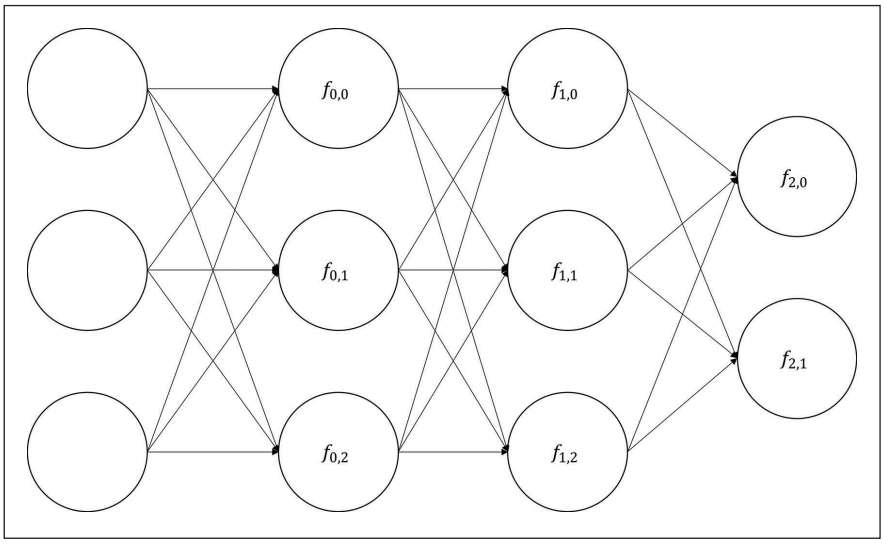

Bild 1: Beispiel eines neuronalen Netzes © CoreAVI

$$f_i = \sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}\right)$$

Bild 2: Lineare Gleichung eines neuronalen Netzes © CoreAVI

Deterministische Inferenzmaschine

Mit diesem Wissen lässt sich nun die Nutzung der Ressourcen von neuronalen Netzen untersuchen. Mit obiger Erklärung kann definiert werden, dass alle Parameter in neuronalen Netzen entweder durch eine Matrix fester Größe oder einen Vektor fester Größe dargestellt werden können. Die Tatsache, dass sie eine feste Größe besitzen, bedeutet, dass der Speicher bekannt ist, der für eine Inferenz mit dem neuronalen Netz vor dessen Ausführung erforderlich ist. Daher ist der erforderliche Speicher deterministisch und lässt sich im Voraus zuweisen.

Zudem ist der Determinismus in der Zeit zu berücksichtigen. Für jeden Knoten gibt es ein Multiplikations- und Additionsverfahren. Bevor die Aktivierungsfunktion angewendet wird, gibt es n Multiplikationen, um das Gewicht mit dem Input zu multiplizieren, $n-1$ Additionen, um alle Gewicht-Input-Multiplikationen zusammenzufassen, und eine weitere Addition, um dem Ergebnis systematische Abweichungen (Bias) hinzuzufügen. Addition und Multiplikation sind grundlegende Vorgänge, und da n eine feste Zahl ist, die auf der Eingabe basiert, lässt sich die Anzahl der Vorgänge, die ausgeführt werden, sowie die Zeit, die zum Abschließen all dieser Vorgänge benötigt wird, deterministisch berechnen.

Für die Aktivierungsfunktion sind weitere Analysen erforderlich um zu beweisen, dass der Vorgang determinis-

$$\sigma(x) = \begin{cases} 0 \; if \; x \leq 0 \\ x \; if \; x > 0 \end{cases}$$

Bild 3: ReLU-Gleichung © CoreAVI

```
Function RELU(x):
    if(x <= 0):
        return 0
    else:
        return x
```

Bild 4: Naive ReLU-Implementierung © CoreAVI

tisch ist. Eine gängige Aktivierungsfunktion, die in vielen neuronalen Netzen verwendet wird, ist die gleichgerichtete lineare Einheit (Rectified Linear Unit, ReLU). Diese wird typischerweise durch die stückweise mathematische Funktion dargestellt (**Bild 3**).

Eine naive Implementierung dieser Funktion zu Darstellungszwecken wäre die Übersetzung der Bedingungen für die Variable x als Verzweigungen (Branches) im Code, einem Pseudocode, wie in **Bild 4**.

Diese Implementierung sieht zunächst deterministisch aus. Analysiert man diese Funktion jedoch, während sie auf einer GPU oder einem anderen parallelen Gerät ausgeführt wird, wird das Branching-Verhalten für den Determinismus problematisch. Auf den meisten modernen GPUs versucht die Hardware, viele Instanzen desselben Vorgangs parallel auszuführen. Aufgrund dieser parallelen Vorgänge müssen deterministische Implementierungen sicherstellen, dass alle Compute Engines der GPU die gleiche Verzweigung im Code nehmen. Im genannten Fall verursacht die Variable x eine Bedingung, bei der jede Recheneinheit einen anderen Pfad basierend auf dem Wert x nehmen muss. Eine andere robustere Möglichkeit zur Implementierung dieses Vorgangs zeigt **Bild 5**.

```
function RELU(x):
    ge_than_zero = (x > 0)
    returnge_than_zero  * x
```

Bild 5: Robustere ReLU-Implementierung © CoreAVI

In dieser Implementierung ist die Verzweigungslogik als Multiplikation mit einem Booleschen Wert dargestellt. Dieser Wert kann entweder 0 oder 1 sein. Ist das Vergleichsergebnis falsch, erhält man also 0 multipliziert mit x, was ebenfalls 0 ist. Ist das Vergleichsergebnis wahr, erhält man 1 multipliziert mit x, was vereinfacht zu x wird. Das implementiert ReLU ohne die Verwendung des Branching. Die Implementierung stellt sicher, dass alle Compute Engines in der GPU denselben Pfad nehmen. Weil das Netzwerk eine feste Anzahl von Knoten hat, ist die Inferenz des Netzwerks deterministisch.

Plattform für neuronale Netze

Die Plattform für ein neuronales Netz bildet der unterstützende Code, der auf einer CPU ausgeführt wird und der es dem neuronalen Netz ermöglicht, auf einem Beschleuniger wie der GPU zu laufen. Bekannte Frameworks wie PyTorch basieren auf Plattformen wie Python, die darauf ausgelegt sind, eine schnelle Entwicklung zu ermöglichen und sich nicht mit den Anforderungen für die ISO-26262-Zertifizierung beschäftigen. Python beispielsweise ist stark von dynamischer Speicherzuweisung und automatischer Speicherverwaltung durch einen Garbage Collector abhängig.

Für autonome Anwendungen wäre es ideal, neuronale Netze mit diesen robusten Tools weiterzuentwickeln und nur auf eine sicherheitskritische Plattform für den Einsatz des neuronalen Netzes in einem sicherheitskritischen System zu wechseln. Die offenen Standards für OpenVX & NNEF von Khronos bieten einen Ansatz, der Entwicklern genau das ermöglicht. Der NNEF-Standard (Neural Network Exchange Format) beinhaltet ein offenes und gemeinsames Format zum Austausch, in das ein neuronales Netz, das in einem Tool wie TensorFlow entwickelt wurde, exportiert und dann in ein anderes Tool importiert werden kann. Das bedeutet, dass TensorFlow-Entwickler ihre neuronalen Netze nehmen und in andere Tools exportieren können.

Der OpenVX-Standard von Khronos liefert eine standardisierte API zur Beschleunigung von Computer-Vision-Anwendungen durch neuronale Netze. Zu dem Standard gehören drei interessante Feature-Sets: Das „Neural Network"-Set, das Funktionen zum Erstellen und Ausführen neuronaler Netze bereitstellt. Außerdem das „NNEF Import"-Set, das die Möglichkeit bietet, neuronale Netze im NNEF-Format in eine OpenVX-Anwendung zu importieren. Und schließlich das „Safety Critical Deployment"-Set, das einen Teil der Features in OpenVX bereitstellt, um den ISO-26262-Zertifizierungsprozess einer Anwendung zu vereinfachen. Diese spezifischen Feature-Sets machen es möglich, OpenVX in autonomen Automobilsystemen zu verwenden.

Schlussfolgerungen

Betrachtet man die Zukunft autonomer Fahrzeuganwendungen wird Sicherheit hier der Schlüssel sein. GPU-beschleunigte Implementierungen neuronaler Netze scheinen eine der besten Möglichkeiten zu bieten, diese autonomen Systeme zu implementieren. Es muss noch mehr Arbeit geleistet werden, um zu beweisen, dass die Absicht eines neuronalen Netzes sicher ist, doch es ist wichtig zu berücksichtigen, dass ein neuronales Netz nicht einfach alleine existiert. Es bedarf viel unterstützender Software, um allein das Deployment eines neuronalen Netzes zu ermöglichen. In einem nach ISO 26262 zertifizierten System ist es entscheidend, alle Komponenten unter Berücksichtigung von Determinismus und Sicherheit zu entwerfen. ■ (eck)

www.coreavi.com

Lucas Fryzek ist Director of Global Field Application Engineers bei CoreAVI. © CoreAVI