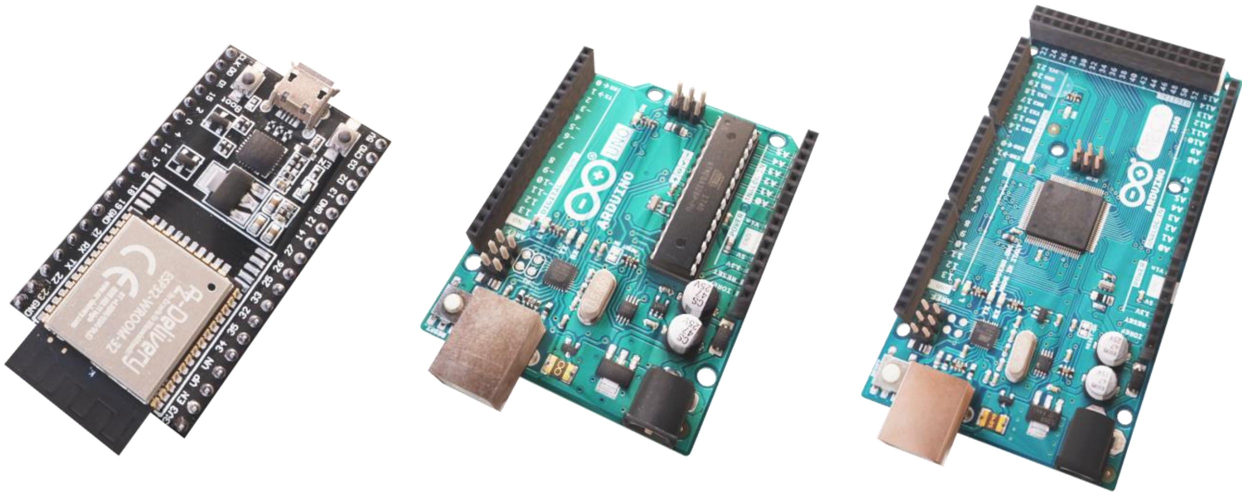
Whitepaper – KI auf Mikrocontrollern

Im Zuge der Vorlesung „Aktuelle Themen“ des Wintersemesters 2021/22
bei Prof. Dr. Andreas Koch
Hochschule der Medien Stuttgart
Leon Hellstern, 42745



Die Nutzung von KI zum Lösen von komplexen Aufgaben, die vorher nur mit dem menschlichen Verstand zu bewältigen waren, hält fortlaufend Einzug in unser tägliches Leben. Dabei wird die Nutzung von neuronalen Netzen ständig in neue Domänen integriert. Besonders interessant ist dabei auch die Nutzung von KI auf Mikrocontrollern, etwa zum Einsatz in Edge Computing Szenarien, bei denen Sensordaten dezentral und lokal ausgewertet werden[2][3]. Dies stellt auf der einen Seite den Gegenentwurf zur Berechnung der Netze auf zentralisierten Servern dar, bei denen das Edge-Gerät nur als Datensammler fungiert. Auf der anderen Seite bedeutet das auch, dass einige neue Herausforderungen entstehen, die zu bewältigende Aufgabe auf einem Mikrocontroller lauffähig zu bekommen.

Dieses Whitepaper beschreibt die zu überwindenden Herausforderungen bei der Optimierung von neuralen Netzen für leistungsbeschränkte Systeme und präsentiert einige effektive Lösungen für genau diese auftretenden Schwierigkeiten anhand eines anschaulichen Beispiels mit gängigen Mikrocontrollern.

Herausforderungen

Im Gegensatz zu gewöhnlichen Rechnersystemen für KI Anwendungen muss man damit rechnen, dass Mikrocontroller über mehrere Größenordnungen weniger an Ressourcen verfügen. Das gilt im Bezug auf die Aufgabe besonders für die in (Tabelle 1) aufgeführten Bereiche, für die als Vergleich ein Espressif ESP32-WROOM32, ein Atmel ATMega328p und ein hypothetischer aktueller Mittelklasse-PC herangezogen wird. Die Zahlen dienen hier nur zur Darstellung der Größenordnungen und sind von verschiedenen Konfigurationseinstellungen abhängig.

Aufgrund dieses massiven Leistungsunterschieds ist es unabdingbar, neuronale Netze für die Nutzung auf Microcontrollern zu optimieren.

	Dezidierter PC	ESP32-WROOM32	ATMega328p
CPU	4 x 3500 MHz (875,0)	2 x 240 MHz (29,2)	1 x 16 MHz (1,0)
Arbeitsspeicher	16384 MB (8192000,0)	520 kB (260,0) (+ 0 – 4 MB via SPI)	2 kB (1,0)
Dauerspeicher	1000 MB (31250,0)	448 kB (14,0) (+ 0 – 4 MB via SPI)	32 kB (1,0)
Bemerkungen	Zusätzliche GPU Beschleunigung!	32bit Controller	8bit Controller

Tabelle 1: Gegenüberstellung eines hypothetischen Mittelklasse - PCs im Vergleich mit üblichen Mikrocontrollern. Zum Ausführen unseres Beispiels fällt auf, dass wir den verfügbaren (Arbeits-)Speicher auf dem PC als "unendlich" betrachten können. Die Werte in Klammern beschreiben den Faktor im Vergleich zum ATMega328p[7][8].

Das Beispiel-Problem: MNIST-Handschrifterkennung

Für die Demonstration von Optimierungsmaßnahmen wird in diesem Whitepaper ein Standardproblem genutzt, welches eine Art „Hello World“ der Künstlichen Intelligenz darstellt: Die Erkennung von Handgeschriebenen Zahlen aus dem MNIST-Datensatz [1]. MNIST steht dabei für „Modified National Institute of Standards and Technology [Database]”, was auf den Ursprungsort des Datensatzes hinweist.

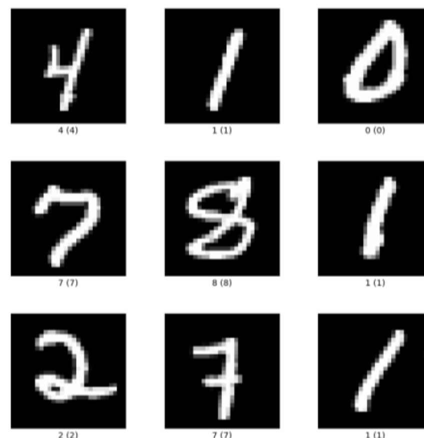


Abbildung 1: Bei der MNIST-Datenbank handelt es sich um Paare von handgeschriebenen Ziffern mit der dazugehörigen Angabe, um welche Zahl es sich dabei handelt[1].

MNIST besteht aus insgesamt 70.000 Paaren von Bildern handgeschriebener Zahlen von 0 bis 9 mit einer Auflösung von je 28 x 28 Pixeln und der dazugehörigen Antwort, um welche Zahl es sich dabei handelt. Die Bilder verfügen nur über Graustufenwerte ohne Farbinformationen. Der Datensatz ist aufgeteilt in 60.000 Bild-Lösungs-Paare zum Trainieren des Netzes und 10.000 Paare zur Verifizierung der aktuellen Erfolgsquote.

Das Netz, durch welches die Aufgabe gelöst werden soll, besteht aus drei voll verknüpften Schichten: Einer Schicht mit 784 Eingabeneuronen (28 x 28), einer versteckten Schicht mit 100 Neuronen und einer Ausgabeschicht, bei der jedes der 10 Neuronen eine Zahl von 0 bis 9 abbildet. Hier wird anhand

einer Aufstellung des Speicherverbrauchs deutlich, dass schon dieses einfache Netz Mikrocontroller an ihre Grenzen bringen kann. Der theoretische Speicherverbrauch im Framework Tensorflow [6] ist in Tabelle 2 aufgeführt.

	Arbeitsspeicher (Bytes)	Dauerspeicher (Bytes)
Neuronen	7152 (894 x 8)	0
Gewichte	635200 (79400 x 8)	0
Gesamt	642352	0

Tabelle 2: Theoretischer Speicherverbrauch des Neuralen Netzes auf einem PC mit Tensorflow [6]. Man beachte, dass eine Fließkommazahl in Python 8 Bytes an Speicher benötigt [9]. Außerdem wird zur Speicherung des Netzes zwischen Ausführungen Dauerspeicher benötigt, was hier vernachlässigt wird.

Beim Vergleich des Speicherverbrauchs mit den von unseren Mikrocontrollern zur Verfügung gestellten Ressourcen ist zu erkennen, dass es beiden nicht möglich ist, dieses neurale Netz ohne Anpassungen im Arbeitsspeicher zu fassen.

Optimierungsmethoden

Exklusion des Netztrainings

Die Tatsache, dass das Trainieren eines Netzes viele Iterationen durch Beispieldaten und eine regelmäßige Anpassung der Netzgewichte erfordert, hat zur Folge, dass ein Aufwands-Ungleichgewicht zwischen Training und Nutzung des Netzes entsteht. Deshalb ist es üblich, das Training auf einem leistungsfähigeren Gerät durchzuführen und die entstehenden Gewichtsdaten danach auf den zu nutzenden Mikrocontroller zu laden. So wird der Controller zu einem reinen Abspielgerät für das Netzwerk, was neben dem Wegfall des Rechenaufwands für das Training auch einen anderen Effekt hat.

Da die Gewichte auf dem Mikrocontroller nicht mehr verändert werden, können diese auf den Dauerspeicher ausgelagert werden, von dem oft entweder mehr zur Verfügung steht, oder welcher sehr einfach als Peripherie an den Controller angebunden werden kann.

In der Speicheraufstellung des MNIST-Beispielproblems in (Tabelle 2) ist erkennbar, dass Gewichtswerte den Großteil der benötigten Speicherressourcen ausmachen. Diese Daten auf den Dauerspeicher zu verladen, kann also je nach Architektur des eingebetteten Systems einen großen Unterschied machen.

Quantisierung von Gewichtsdaten und Aktivierungen

Da Gewichtsdaten und Aktivierungen einen großen Teil des Speicherverbrauchs eines Netzes ausmachen, spielt auch der Datentyp dieser Werte eine wichtige Rolle. Durch die Reduzierung der Genauigkeit der einzelnen Werte kann demnach der Verbrauch drastisch verringert werden.

Bei einer 8-bit Quantisierung der ursprünglich als Double-Fließkommazahl abgespeicherten Gewichte wird der Wertebereich der Daten pro Netzschicht in 256 Bereiche eingeteilt und die einzelnen Werte an den am nächsten liegenden Bereichswert gezogen. So kann im Vergleich zum ursprünglichen Tensorflow-Netz der Speicherbedarf auf ein Achtel reduziert werden.

In den meisten Fällen hat eine moderate Quantisierung der Gewichts- und Aktivierungswerte keinen nennenswerten Einfluss auf die Genauigkeit der Ergebnisse. Das zeigen Choi et al. [10] in ihrer Veröffentlichung „Towards the Limit of Neural Network Quantization". Erst ab einer Wortbreite unter 6bit wird die Genauigkeit des Netzes bei einer gleichförmigen Quantisierung der Werte bedeutend schlechter als die der Referenz. Beim MNIST-Beispielnetzwerk in Tensorflow hat eine 8bit Quantisierung der Werte eine Verschlechterung der Treffsicherheit von unter 0,1% zur Folge[12].

(Abbildung 2) zeigt schematisch, wie eine Quantisierung die Gewichts- und Aktivierungswerte verändert.

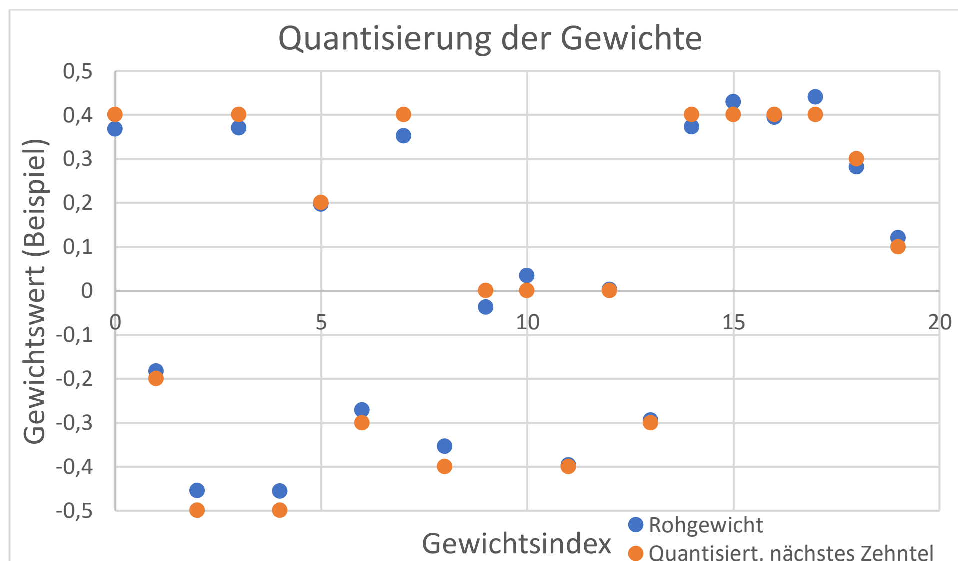


Abbildung 2: Schematische Darstellung einer Quantisierung. In diesem Fall wird der Wertebereich in 11 Bereiche eingeteilt, die jeweils ein Zehntel auseinanderliegen. Die Rohwerte werden dann an den nächsten Quantisierungswert gezogen.

Eine Möglichkeit, den Genauigkeitsabfall weiter zu verringern, ist, das Training ebenfalls auf dem Quantisierten Netz durchzuführen. Dieses Verfahren nennt man „quantisierungsbewusstes Training" [5]. Beim Trainieren des Netzes muss demnach eine Mindestschrittweite eingehalten werden, damit überhaupt eine Änderung der Gewichte möglich ist. Ebenfalls kann es für die Ergebnisse des Netzwerks zuträglich sein, die Aktivierungen auf eine höhere Wortbreite zu quantisieren. Diese Änderung schlägt sich nicht so sehr im Gesamtspeicherverbrauch nieder wie die Verkleinerung der Gewichtswerte, kann allerdings in einigen Fällen zu einer deutlichen Verbesserung der Ergebnisse führen.

Man sollte bei der Wahl der Quantisierungsmethode darauf achten, dass dadurch möglicherweise vorher von der verwendeten Hardware zur Verfügung gestellte Beschleunigungsmechanismen auf Chipbasis ausgehebelt werden können. Dadurch wird die Berechnungsgeschwindigkeit deutlich verringert. Ein Beispiel dafür sind SIMD – Anweisungen für Prozessoren (Single Instruction Multiple Data), die nur bei bestimmten Wortbreiten und Datentypen aktiv werden können, um parallel mehrere Berechnungen durchzuführen [11].

Bereinigung des Netzes und Clustering

Wenn nicht der Speicherverbrauch des Netzes zur Ausführungszeit, sondern die Menge an zu transferierenden Daten bei einem Netzdownload eine Herausforderung darstellt, kann man dem mithilfe einer Bereinigung entgegenwirken. Dabei werden alle Gewichtswerte, die einen vernachlässigbaren Einfluss auf das Endergebnis des Netzes haben, auf 0 gesetzt. Während der

Ausführung ist dabei kein Unterschied zu erkennen, aber das Netz lässt sich dadurch viel besser komprimieren [17].

Eine weitere Art, die Komprimierbarkeit Neuronaler Netze zu erhöhen, ist, die Anzahl der einzigartigen Gewichtswerte zu verringern. Dafür verbindet man Gewichte mit sehr ähnlichen Werten zu Clustern, die sich nach der Verarbeitung einen gemeinsamen Mittelwert teilen. Dadurch entsteht eine Tabelle an Clustergewichten und den dazugehörigen Gewichtsindizes, die die Verbindung der Gewichte zu ihrem Cluster zeigen. Mithilfe dieser Methode kann die Größe des komprimierten Netzes regelmäßig halbiert werden [13].

Anpassung des Problems

Möchte man Neuronale Netze auf möglichst leichtgewichtige Systeme portieren, sind womöglich alle vorher genannten Optimierungen nicht ausreichend, um die Aufgabe durchführen zu können. Als weitere Maßnahme sollte also in Betracht gezogen werden, die zu Grunde liegende Aufgabe bei möglichst gleichbleibender Genauigkeit des Netzes zu vereinfachen.

Eine effektive Möglichkeit beim hier genutzten MNIST-Beispiel ist, die Auflösung des Bildes und damit die Größe des Eingangsvektors zu verkleinern. Nutzt man zur Erkennung der Ziffern ein Eingangsbild von 14 x 14 statt 28 x 28 Pixeln, verringert sich die Anzahl der Eingangswerte und die der Gewichte des Netzes zwischen Schicht 1 und 2 um 75%. Diese Änderung hat Einfluss auf den Speicherverbrauch während der Laufzeit, die Größe des Netzes beim Transfer und die Geschwindigkeit der Ausführung.

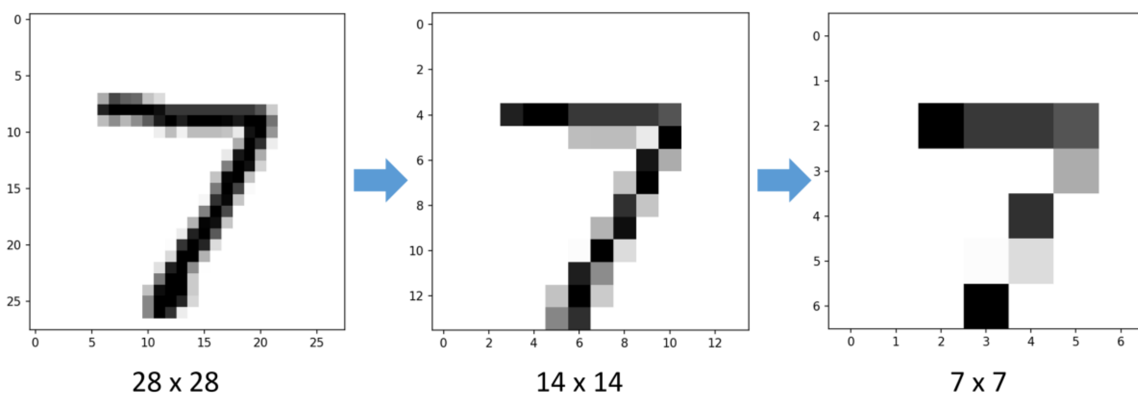


Abbildung 3: Selbst bei einer Verkleinerung des Eingangsvektors zu 14 x 14 Pixeln ist diese Zahl des MNIST-Datensatzes noch gut zu erkennen. Bei einer zu starken Verkleinerung leidet allerdings die genaue Unterscheidung von optisch ähnlichen Zahlen, wie zum Beispiel 1 und 7 oder 3 und 9.

Solange die Anpassung des Problems moderat gehalten wird, kann man auch damit rechnen, eine ähnliche Genauigkeit des Neuronalen Netzes beizubehalten. In diesem Fall sinkt diese nach 10 Trainingsepochen um lediglich 1,2% (Abbildung 4). Eine Verkleinerung des Inputs auf 7 x 7 Pixel hat allerdings eine Verringerung der Genauigkeit von etwa 20% zur Folge, was in den meisten Fällen als nicht tragbarer Verlust gilt.

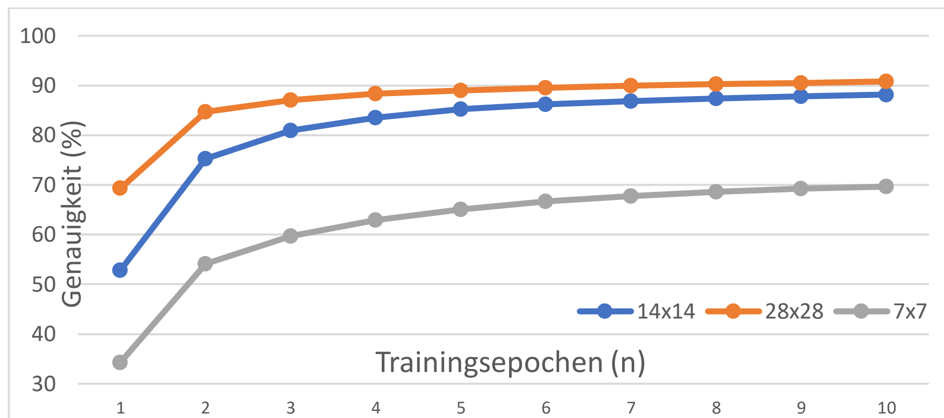


Abbildung 4: Unterschiede in der Genauigkeit zwischen dem MNIST-Beispielnetz mit einem Eingangsvektor von 7 x 7, 14 x 14 und 28 x 28 Pixeln über mehrere Trainingsepochen hinweg.

Auch möglich ist im Fall der MNIST-Aufgabe die Konvertierung des Eingangsbildes in ein binäres Schwarz-Weiß-Bild, mit der der Eingangsvektor bei gleichbleibender Auflösung im Speicherbedarf verkleinert wird. Das hat allerdings zur Folge, dass zur Laufzeit ein Entpacken des Wertes stattfinden muss und dementsprechend die Ausführungsgeschwindigkeit darunter leidet.

Hardwareanpassungen

Sollten bis hier alle vorangegangenen Optimierungsversuche gescheitert sein, ist eine Re-Evaluation der Hardwaremöglichkeiten sinnvoll. Dafür können mehrere Ansätze in Betracht gezogen werden. Mikrocomputer (zum Beispiel eine Raspberry Pi Plattform) bieten in Vergleich zu Mikrocontrollern deutlich mehr Ressourcen und können weiterhin mit einem moderaten Stromverbrauch, einem kleinen Formfaktor und relativ geringen Stückkosten in das gegebene Problemprofil passen.

Ein weit kostenintensiverer Ansatz ist das Entwickeln von individueller Hardware zur Lösung von Performanceproblemen in Embedded Systemen. So nutzt Google seit 2017 erfolgreich eigens zur Berechnung von neuronalen Netzen entwickelte Beschleunigungskarten in ihren Datencentern, welche im Vergleich zu GPUs eine enorm verbesserte Performance pro Watt Leistungsaufnahme liefern [14]. Diese Maßnahme ist allerdings durch hohe Entwicklungs- und Herstellungskosten für viele Unternehmen nicht erreichbar.

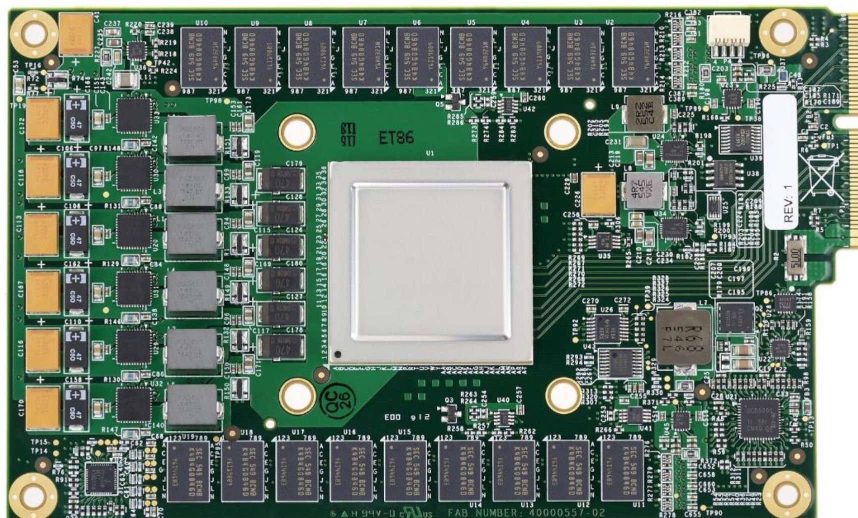


Abbildung 5: Die erste Generation von dezidierten Beschleunigern für Neuronale Netze von Google wurden als PCI-Steckkarte in bestehende Rechnersysteme eingebunden und liefern 29 mal mehr Rechenleistung pro Watt im Vergleich zu aktuellen GPUs [15].

Zu guter Letzt ist es sinnvoll, auch das klassische Modell der Datenverarbeitung in Datencentern in Betracht zu ziehen. Sollte das verwendete Neuronale Netz so komplex sein, dass Edge Computing aktuell keine valide Option mehr darstellt, kann man so auf eine gut skalierbare Alternative zurückgreifen, die allerdings von einer stabilen Netzwerkverbindung abhängig ist.

Effektivität – MNIST auf dem AtMega328p

Um die Effektivität der hier genannten Maßnahmen zu demonstrieren, wird das MNIST-Beispielnetz für die Nutzung auf dem AtMega328p Mikrocontroller optimiert. Um das zu bewerkstelligen, wird zu allererst auf eine 8bit Quantisierung aller im Netz befindlichen Werte zurückgegriffen, was den Speicherverbrauch auf ein Achtel reduziert. Des Weiteren wird ein Lernen des Netzes auf dem Mikrocontroller exkludiert, um die am PC vortrainierten Gewichte auf den beim AtMega328p deutlich größeren Dauerspeicher zu verschieben.

Selbst nach diesen Schritten würden die Gewichte des Netzwerks 80kB an Speicher benötigen, auf dem Controller stehen allerdings nur 32kB zur Verfügung. Deshalb wird auf eine sinnvolle Verkleinerung der Aufgabe zurückgegriffen und der Inputvektor von 28 x 28 auf 14 x 14 Pixeln verkleinert. Dadurch verkleinert sich die Größe der Gewichte zwischen der Input- und der versteckten Schicht auf ein Viertel des Ursprungswertes und so passt das gesamte Netz auf den Mikrocontroller und die Optimierung ist erfolgreich abgeschlossen. Wir mussten hier nicht auf die Nutzung von externer- oder eigens entwickelter Zusatzhardware zurückgreifen. Die KI-Aufgabe wird mit einer zum Original sehr vergleichbaren Genauigkeit ausgeführt (1,8% Genauigkeitsverlust mit 10 Epochen Training). In (Abbildung 6) ist der letztendliche Speicherverbrauch aufgestellt.

	Arbeitsspeicher (Bytes)	Dauerspeicher (Bytes)
Neuronen	306 (306 x 8bit)	0
Gewichte	0	20600 (20600 x 8bit)
Gesamt	306	20600
Vergl. PC	642352	0

Abbildung 6: Speicheraufstellung unseres für den AtMega328p optimierten neuronalen Netzes zur Bewältigung der MNIST-Beispielaufgabe im Vergleich mit dem PC.

Fazit

Mit den in diesem Whitepaper vorgestellten Verfahren zur Optimierung von neuronalen Netzen ist es durchaus möglich, relevante KI Anwendungen auf Low-Spec, Low-Power und Low-Cost Geräten sinnvoll auszuführen. Das zeigt das hier vorgestellte Beispiel, bei dem die MNIST Handschrifterkennung erfolgreich auf einem extrem beschränkten 8bit Mikrocontroller mit 2kB Arbeitsspeicher und 32kB Dauerspeicher durchgeführt werden konnte. Dabei mussten kaum Einbußen hinsichtlich der Genauigkeit der Ergebnisse in Kauf genommen werden.

Die Anwendung dieser Verfahren ist einfach umzusetzen und liefert schnell gute Ergebnisse. So sollte es auch kleineren oder unerfahrenen Teams möglich sein, KI Techniken auf beschränkter Hardware umzusetzen.

Quellen

Alle Quellen wurden am 22.03.2022 zuletzt auf Aktualität überprüft

[1] THE MNIST DATABASE of handwritten digits
http://yann.lecun.com/exdb/mnist/
[2] Industry of Things: Was bedeutet Edge Computing?
https://www.industry-of-things.de/wasbedeutet-edge-computing-a-678225/
[3] IBM: Was ist Edge Computing?
https://www.ibm.com/de-de/cloud/what-is-edge-computing
[4] KI auf Microcontrollern: Ein Beitrag von Renesas Electronics
https://www.elektroniknet.de/halbleiter/mikrocontroller/ki-auf-mikrocontrollern.159193.html
[5] Modelloptimierungsmethoden und ihre praktische Anwendung in Tensorflow
https://www.tensorflow.org/lite/performance/model_optimization
[6] Tensorflow Webseite und Ressourcen
https://www.tensorflow.org/
[7] ESP32 Wroom 32 Datenblatt
https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf
[8] Atmega328p Datenblatt
https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf
[9] Python Floating Point Informationen
https://www.pythontutorial.net/advanced-python/python-float/
[10] Y. Choi, M. El-Khamy, and J. Lee: Towards the Limit of Neural Network Quantization, ICLR 2017
https://arxiv.org/pdf/1612.01543.pdf
[11] The Basics of SIMD Programming (im Kontext der Cell Prozessorarchitektur)
http://ftp.cvut.cz/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html
[12] Post Training Integer Quantization in Tensorflow
https://www.tensorflow.org/lite/performance/post_training_integer_quant
[13] Clustering in Tensorflow – Results and Performance
https://www.tensorflow.org/model_optimization/guide/clustering#results
[14] Norman P. Jouppi et al: In-Datacenter Performance Analysis of a Tensor Processing Unit, 2017
https://arxiv.org/ftp/arxiv/papers/1704/1704.04760.pdf
[15] An in-depth look at Googles first tensor processing unit
https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu
[16] M. Nagel et al.: A White Paper on Neural Network Quantization, Qualcomm AI Research, 2021
https://arxiv.org/pdf/2106.08295.pdf
[17] Tensorflow: Pruning
https://www.tensorflow.org/model_optimization/guide/pruning

Weitere Quellen für die Durchführung von Experimenten und die Erstellung der dazugehörigen Präsentation

Alle Quellen wurden am 22.03.2022 zuletzt auf Aktualität überprüft

https://www.tensorflow.org/datasets/catalog/mnist
https://www.tensorflow.org/lite/microcontrollers?hl=en
https://www.tensorflow.org/lite/microcontrollers/build_convert?hl=en
https://www.tensorflow.org/lite/performance/post_training_quantization?hl=en
https://www.nongnu.org/avr-libc/user-manual/group__avr__pgmspace.html
https://store.arduino.cc/products/arduino-uno-rev3/
https://store.arduino.cc/products/arduino-mega-2560-rev3/
https://www.mouser.de/datasheet/2/891/esp32-wroom-32d_esp32-wroom-32u_datasheet_en-1365844.pdf
https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf
https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf
https://www.tinyml.org/about/
https://developers.google.com/machine-learning/glossary
https://keras.io/api/
https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/quickstart/beginner.ipynb#scrollTo=OeOrNdnkEEcR
https://www.youtube.com/watch?v=iTj0lcVSIVU
https://www.automate.org/industry-insights/getting-started-with-ai-based-predictive-maintenance