AI-ANNE: (A) (N)eural (N)et for (E)xploration

Transferring Deep Learning Models onto Microcontrollers and Embedded Systems

a Working Paper and Manual by Dennis Klinkhammer

Abstract (ENG)

This working paper explores the integration of neural networks onto resource-constrained embedded systems like a Raspberry Pi Pico / Raspberry Pi Pico 2. A TinyML aproach transfers neural networks directly on these microcontrollers, enabling real-time, low-latency, and energy-efficient inference while maintaining data privacy. Therefore, AI-ANNE: (A) (N)eural (N)et for (E)xploration will be presented, which facilitates the transfer of pre-trained models from high-performance platforms like TensorFlow and Keras onto microcontrollers, using a lightweight programming language like MicroPython. This approach demonstrates how neural network architectures, such as neurons, layers, density and activation functions can be implemented in MicroPython in order to deal with the computational limitations of embedded systems. Based on the Raspberry Pi Pico / Raspberry Pi Pico 2, two different neural networks on microcontrollers are presented for an example of data classification. As an further application example, such a microcontroller can be used for condition monitoring, where immediate corrective measures are triggered on the basis of sensor data. Overall, this working paper presents a very easy-to-implement way of using neural networks on energy-efficient devices such as microcontrollers. This makes AI-ANNE: (A) (N)eural (N)et for (E)xploration not only suited for practical use, but also as an educational tool with clear insights into how neural networks operate.

Abstract (GER)

Dieser vorläufige Artikel befasst sich mit der Integration neuronaler Netze in ressourcenlimitierte und eingebettete Systeme wie den Raspberry Pi Pico / Raspberry Pi Pico 2. Ein TinyML-Ansatz überträgt dabei neuronale Netze direkt auf einen Mikrocontroller und ermöglicht so eine latenzarme und energieeffiziente Datenanalyse bei gleichzeitiger Sicherung sensibler Daten. Zu diesem Zweck wird KI-ENNA: (E)in (N)euronales (N)etz zum (A)usprobieren vorgestellt, das die Übertragung von vortrainierten und rechenintensiven Modellen auf Basis von TensorFlow und Keras auf Mikrocontroller unter Verwendung einer leichtgewichtigen Programmiersprache wie MicroPython ermöglicht. Dieser Ansatz verdeutlicht dabei, wie die den neuronalen Netzen zugrundeliegende Architektur in Form von Neuronen, Schichten, Dichte und Aktivierungsfunktionen in MicroPython implementiert werden kann, um unter den Ressourcenlimitationen von eingebetteten Systemen funktionsfähig zu sein. Auf Grundlage des Raspberry Pi Pico / Raspberry Pi Pico 2 werden zwei verschiedene neuronale Netze auf Mikrocontrollern für ein Beispiel zur Datenklassifizierung vorgestellt. In der Praxis wäre mit einem solchen Microcontroller darüber hinaus eine Zustandsüberwachung möglich, bei denen auf Basis von Sensordaten sofortige Korrekturmaßnahmen ausgelöst werden. Dadurch stellt dieser vorläufige Beitrag insgesamt eine sehr leicht umzusetzende Möglichkeit vor, wie neuronale Netze auf energieeffizienten Geräten wie Mikrocontroller zur Anwendung gebracht werden können. Dadurch ist KI-ENNA: (E)in (N)euronales (N)etz zum (A)usprobieren nicht nur eine Option für die Praxis, sondern gleichermaßen ein didaktisches Tool mit anschaulichen Einblicken in die Funkktionsweise neuronaler Netze.

Keywords

TinyML, EdgeAI, Microcontroller, Embedded Systems, Machine Learning, Deep Learning

(I) Introduction

Artificial Intelligence for Microcontrollers

Machine Learning and Deep Learning are increasingly driving innovation across various fields (LeCun et al. 2015), with a notable expansion into embedded systems, bringing their capabilities closer to data sources. This trend, known as TinyML, is especially prominent in microcontrollers and Internet of Things devices. TinyML offers several advantages over cloud-based artificial intelligence, including improved data privacy, lower processing latency, energy efficiency, and reduced dependency on connectivity (Ray 2022). One key application of TinyML is in condition monitoring, where neural networks can be used to detect anomalies directly within sensors so that immediate corrective actions can be taken automatically (Cioffi et al. 2020). Therefore, when these microcontrollers are connected to microscopes in medical diagnostics or machines for industrial production, they are referred to as embedded systems.

While many high-performance frameworks like TensorFlow and Keras for Python are designed for powerful hardware such as GPUs, these frameworks are unsuitable for embedded systems due to their limited computational resources (Delnevo et al. 2023). To address this issue, the architecture of neural networks need to be reconstructed in a leightweight programming language like MicroPython. Thus, neural networks trained on high-performance hardware can be transferred onto resource-constrained devices like microcontrollers while achieving the same outputs and results (Ray 2022). However, training or developing neural networks directly on embedded systems remains a complex challenge. AIfES: (A)rtificial (I)ntelligence (f)or (E)mbedded (S)ystems, as presented below as a comparable approach, has shown initial success here (Wulfert et al. 2024).

This working paper presents AI-ANNE: (A) (N)eural (N)et for (E)xploration, a new method for transferring pre-trained neural networks onto a microcontroller. For this purpose, neurons, layers, density and activation functions as the underlying foundation of neural networks were coded in MicroPython in order to be able to reconstruct the original neural networks trained with TensorFlow and Keras. How the transfer works and a suitable application example are part of this working paper. The underlying foundation of neural networks and their counterparts in MicroPython are presented successively.

Comparable Approaches and Added Value

AIfES: (A)rtificial (I)ntelligence (f)or (E)mbedded (S)ystems is a flexible software framework designed by Fraunhofer to run deep learning models on small, low-power devices like microcontrollers (Wulfert et al. 2024). It simplifies the process of building, training, and running models directly on these devices, without needing powerful external systems. Users can customize the framework to fit their needs by choosing different model components, like types of layers or how data is processed. AIfES: (A)rtificial (I)ntelligence (f)or (E)mbedded (S)ystems can run pre-trained models and train new ones on the device itself, saving energy and protecting privacy (Wulfert et al. 2024). It outperforms similar tools in terms of speed and memory usage for certain tasks. Future improvements will focus on making it even more efficient and supporting new types of deep learning models.

AI-ANNE: (A) (N)eural (N)et for (E)xploration is a similar approach and, as an open framework, enables the flexible expansion of the underlying activation functions in order to explore their performance while simultaneously the number of neurons and layers can be adjusted easily in MicroPython. This flexibility can also be used for fine-tuning directly on the microcontroller. As a result, AI-ANNE: (A) (N)eural (N)et for (E)xploration allows the learning behavior of the neural networks to be observed and creates a didactic value for its users. Two neural networks are already pre-installed: One with six neurons in a total of three layers and one with eight neurons in a total of four layers. In the given example, the learning behavior of the various neural networks can be investigated with the pre-installed IRIS dataset. The transparent insight into the MicroPython code also opens up didactic application potential. The interaction with the microcontroller takes place via Thonny. All the necessary components are presented below.

(II) Requirements

Raspberry Pi Pico / Raspberry Pi Pico 2

The Raspberry Pi Pico is powered by the RP2040 microcontroller, which features a dual-core ARM Cortex-M0+ processor running at 133 MHz, with 264 KB on-chip SRAM and 2 MB onboard flash memory. It offers a wide range of connectivity options, including USB for power and data transfer, up to two I2C, SPI, and UART interfaces for communication, as well as 16 PWM channels for precise control of external devices.

The board also includes three 12-bit ADC channels for analog input, and it supports a range of peripherals such as a real-time clock (RTC), timers, and interrupt handling through a nested vectored interrupt controller (NVIC). For power, the Raspberry Pi Pico operates on a voltage range of 1.8V to 3.6V, with typical consumption between 20-100 mA, and can be powered via the micro-USB port or the VSYS pin for external power sources like batteries or regulated supplies.

In 2024 an updated Raspberry Pi Pico 2 was introduced, which is powered by the RP2350 microcontroller, which features a dual-core ARM Cortex-M33 processor running at 150MHz and 520 KB on-chip SRAM and 4 MB onboard flash memory. Both can be operated with MicroPython. It can be assumed that the Raspberry Pi Pico 2 can calculate larger datasets and more complex neural networks with AI-ANNE: (A) (N)eural (N)et for (E)xploration. Therefore, the use of a Raspberry Pi Pico 2 is recommended for practical use; The use of the Raspberry Pi Pico might be sufficient for educational purpose.

MicroPython

MicroPython is an efficient, lightweight implementation of the Python programming language designed to run on microcontrollers and embedded systems with constrained resources (Delnevo et al. 2023). Unlike the full Python environment, MicroPython is optimized to operate within the memory and processing limits typical of small-scale devices, offering a streamlined interpreter and a subset of Python's standard libraries. MicroPython retains much of Python's high-level syntax and ease of use, making it accessible to developers familiar with Python. It is particularly well-suited for rapid prototyping, development, and deployment of machine learning and neural network models on embedded platforms, where resources such as memory, computational power, and storage are limited.

In the context of neural network applications, MicroPython is often used in edge computing scenarios, where deep learning models need to be deployed directly onto microcontroller-based systems for real-time, localized inference. Although MicroPython does not natively support the extensive numerical libraries found in full Python (e.g., TensorFlow and Keras), it is possible to reproduce the basic architecture of neural networks with AI-ANNE: (A) (N)eural (N)et for (E)xploration in MicroPython from scratch.

Thonny

Thonny is a simple, user-friendly program that works on all major computer systems. It makes it easy to connect with and program the Raspberry Pi Pico / Raspberry Pi Pico 2. Thonny enables users to quickly write and test code, manage files and fix any mistakes with helpful tools. It can be described as a tool for those just starting with MicroPython on microcontrollers and embedded systems. With Thonny, AI-ANNE: (A) (N)eural (N)et for (E)xploration can simply be flashed onto the microcontroller.

Thonny can be downloaded here: https://thonny.org/

(III) Basic Knowledge

Architecture of Neural Networks

A neural network is a computational model inspired by the way biological neural systems process information. It consists of interconnected layers of nodes, or neurons, which transform input data into output predictions through a series of mathematical operations (LeCun et al. 2015). In the context of implementing neural networks in MicroPython, the architecture must be designed to operate within the resource constraints of embedded systems (Ray 2022). Despite these constraints, the basic components of a neural network - neurons, layers, density, and activation functions - can still be effectively modeled (Sakr et al. 2021).

a) Neurons

A neuron in a neural network is a computational unit that receives inputs, applies a weight to each input, sums the weighted inputs, and passes the result through an activation function to produce an output. In MicroPython (Appendix - C), each neuron can be represented as a mathematical operation involving inputs and weights, with the output being calculated via simple matrix operations.

The general form of the neuron's computation can be expressed with y as the output of the neuron, f as the activation function, x_i as the inputs, w_i as the corresponding weights and b as the bias:

$$y = f(\sum_i w_i x_i + b)$$

In a MicroPython implementation, these calculations can be done using basic array operations, making it computationally efficient for small-scale neural networks on microcontrollers.

b) Layers

A neural network is typically structured as a series of layers, where each layer consists of multiple neurons. There are three main types of layers in a typical neural network architecture:

- Input Layer: This is the first layer of the network and receives the raw input data. Each neuron in the input layer corresponds to a feature in the input data. With AI-ANNE: (A) (N)eural (N)et for (E)xploration, this layer would typically read sensor values or external data directly.
- Hidden Layers: Between the input and output layers, the network may contain one or more hidden layers. These layers are responsible for learning complex representations of the input data. Each hidden layer is composed of neurons that perform weighted sums of the outputs of the previous layer, followed by an activation function.
- Output Layer: This is the final layer of the network, which produces the prediction or classification result. The output layer's structure depends on the problem being solved (e.g., binary classification, multi-class classification, logistic regression).

MicroPython can represent the layers as a list of lists (or arrays) of neurons, where each element stores the weights, biases, and outputs of the neurons in the respective layer.

c) Density

The density of a layer refers to the number of neurons in that layer. In a fully connected (dense) layer, each neuron in the current layer is connected to all neurons in the previous layer. This is the most common configuration in neural networks. For example: In a dense layer with 5 neurons, each neuron in the layer receives input from all neurons in the previous layer, and the layer will contain 5 sets of weights and biases to be learned during training. MicroPython can implement a dense layer efficiently using matrix multiplication (Appendix - A, all of a sudden, school math seems very practical!). Each layer's input and weight matrices are multiplied together, followed by the addition of a bias term, and the result is passed through an activation function.

d) Activation Functions

An activation function determines whether a neuron should activate, so that the neuron's output will be passed forward to the next layer. The activation function introduces non-linearity into the network, allowing it to learn complex patterns in the data. Common activation functions include:

- Sigmoid: The Sigmoid function outputs values between 0 and 1 and is often used in binary classification tasks.

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- ReLU (Rectified Linear Unit): ReLU outputs the input directly if it is positive; otherwise, it outputs zero. It is widely used in hidden layers due to its simplicity and efficiency in preventing vanishing gradients.

$$ReLU(x) = \max(0, x)$$

- Leaky ReLU: The Leaky ReLU activation function is a variant of the traditional ReLU activation function, which addresses a common issue known as the "dying ReLU" problem. This problem occurs when neurons become inactive and stop learning because their output is always zero (when the input is negative). Leaky ReLU overcomes this by allowing a small, non-zero gradient when the input is negative. The key difference between ReLU and Leaky ReLU is that when $x \leq 0$, instead of the output being zero (as in ReLU), the output is a small negative value. The parameter *alpha* controls the slope of this negative region, typically with small values such as 0.01 or 0.1.

$$LeakyReLU(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

- Tanh: The hyperbolic tangent function Tanh outputs values between -1 and 1. It is similar to the sigmoid but has a wider output range, making it useful for some applications where the network needs to model both positive and negative values.

$$Tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- Softmax: Often used in the output layer of classification networks, the Softmax function normalizes the output to produce a probability distribution across multiple classes.

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

In MicroPython, these activation functions can be implemented as simple functions, leveraging MicroPython's built-in math library (Appendix - B). Given the computational constraints of embedded devices, it is crucial to choose lightweight activation functions like ReLU, which avoid the more computationally expensive exponentiation operations used in Sigmoid or Tanh. AI-ANNE: (A) (N)eural (N)et for (E)xploration passes data through the neural network layer by layer, and each neuron's output is computed based on the inputs and weights. In a MicroPython environment, due to the limited processing power and memory, a smaller network with fewer layers and neurons might be preferred, and training could be done in advance on a more powerful system before deployment to the embedded device.

e) Weights and Biases

In neural networks, weights and biases are key parameters that the model learns during the training process to make accurate predictions.

Weights are the parameters that scale the input values as they pass through the network. They determine the strength of the connections between neurons in adjacent layers. Each connection between two neurons has its own weight, and the value of this weight influences how much the input from the previous layer affects the output of the current layer. In mathematical terms, if a neuron receives an input x and its corresponding weight is w, the contribution of that input to the neuron's output is $w * x$. Weights are adjusted during training using optimization techniques, such as gradient descent, to minimize the model's prediction error.

Biases, on the other hand, allow the model to shift the output independently of the weighted sum of inputs. They are added to the weighted sum before it is passed through the activation function. This allows the network to better model complex relationships in the data by shifting the activation function's threshold. For example, if the weighted sum of inputs to a neuron is represented as $z = w * x + b$, the bias term b shifts the output, enabling the model to learn more effectively. Like the weights, biases are also learned during training.

Confusion Matrix

A confusion matrix is like a scoreboard that indicates how well a neural network is doing at making predictions. It's especially useful for classification problems, where the goal is to assign items to categories. For the purpose of binary classification a confusion matrix can be a table with two rows and two columns. Each cell tells something about the model's predictions compared to the actual results, which can be interpreted as follows:

- True Positives (TP): The model correctly predicted the positive class.
- True Negatives (TN): The model correctly predicted the negative class.
- False Positives (FP): The model incorrectly predicted the positive class (Type I Error).
- False Negatives (FN): The model incorrectly predicted the negative class (Type II Error).

The Accuracy of the neural network based prediction can be calculated as a metric based on this. Accuracy as a percentage is a metric that provides information about how often the model is right overall. Below is an illustrative example (Tab. 1) of a confusion matrix with actual values in the rows and predicted values in the columns:

(Tab. 1) Confusion Matrix Example

	Predicted Positive	Predicted Negative
Actual Positive	09	01
Actual Negative	00	10

In this binary classification, 09 of the positive class are classified correctly and 01 incorrectly (Type II Error). Accordingly, 10 of the negative class are classified correctly and 00 incorrectly. This results in an Accuracy of 95% (19/20 = 0.95) and is based on the second application example of a neural network with 6 neurons, which is presented below. In addition to this metric, there are other metrics for the evaluation of neural networks, e.g. Precision, Recall and F1 Score. For this example and as basic introduction to AI-ANNE: (A)(N)eural (N)et for (E)xploration, the focus on accuracy shall suffice.

(IV) Application Example

Binary Classsification

The IRIS dataset is a widely recognized dataset in statistics, machine learning and deep learning and often used for classification problems. Introduced by the British biologist and statistician Ronald A. Fisher in 1936, it contains 150 instances of iris flowers, each with four attributes that describe their physical characteristics. These attributes include the sepal length, sepal width, petal length, and petal width, all measured in centimeters.

Based on these four attributes, the objective of the dataset is to classify each flower into one of three species: Setosa, Versicolor, and Virginica. The IRIS dataset serves as an ideal example for demonstrating and testing machine learning algorithms and neural networks as deep learning models, particularly for classification tasks. Its manageable size, clear distinctions between classes and straightforward nature make it a popular choice for exploring classification techniques. However, classifying some of the species, particularly Versicolor and Virginica, can be challenging due to their overlapping characteristics, making the task more complex for certain machine learning algorithms and neural networks as deep learning models.

In the pre-installed example, AI-ANNE: (A) (N)eural (N)et for (E)xploration was entrusted with the classification of Versicolor and Virginica accordingly. The result will be a percentage, as presented before as Accuracy.

Solution with 8 Neurons

First, with TensorFlow and Keras pre-trained weights and biases must be transferred onto the microcontroller. The weights and biases of the neural network with 8 neurons need to be coded in a specific order in MicroPython (Tab. 2). The first layer, the so called input layer, contains two neurons that process the input from the four independent variables of the IRIS dataset. The weights $w1$ are coded accordingly with two columns and four rows in MicroPython. The two neurons are correspondingly provided with two biases $b1$. The weights and biases for the number of neurons in the other layers are coded accordingly: Three neurons follow in the first hidden layer, followed by two neurons in the second hidden layer and one neuron in the output layer.

(Tab. 2) Weights and Biases for 8 Neurons

```
w1 = [[-0.75323504, -0.25906014],
      [-0.46379513, -0.5019245 ],
      [ 2.1273055 ,  1.7724446 ],
      [ 1.1853403 ,  0.88468695]]
b1 = [0.53405946, 0.32578036]
w2 = [[-1.6785783,  2.0158117,  1.2769054],
      [-1.4055765,  0.6828738,  1.5902631]]
b2 = [ 1.18362  , -1.1555661, -1.0966455]
w3 = [[ 0.729278  , -1.0240695 ],
      [-0.80972326,  1.4383037 ],
      [-0.90892404,  1.6760625 ]]
b3 = [0.10695826, 0.01635581]
w4 = [[-0.2019448],
      [ 1.5772797]]
b4 = [-1.2177287]
```

For pre-training in TensorFlow and Keras the ReLU activation function was used in the input layer, followed by Tanh in the first hidden layer, Softmax in the second hidden layer and Sigmoid in the output layer. In MicroPython (Tab. 3), where users have the option to change the number of neurons, layers as well as changing the activation functions used, the corresponding code would be:

(Tab. 3) Neural Network with 8 Neurons

```
yout1 = dense(2, transpose(Xtest), w1, b1, 'relu')
yout2 = dense(3, yout1, w2, b2, 'tanh')
yout3 = dense(2, yout2, w3, b3, 'softmax')
ypred = dense(1, yout3, w4, b4,'sigmoid')
```

The Accuracy of this neural network with 8 neurons is 90%. A variation of the number of neurons, layers as well as activation functions with their weights and biases can influence the accuracy accordingly. Therefore, AI-ANNE: (A) (N)eural (N)et for (E)xploration enables direct customization of the code in MicroPython via Thonny.

Solution with 6 Neurons

The next neural network differs in the number of neurons, the layers and the activation functions used. This time there are three neurons in the input layer (Tab. 4). Accordingly, the weights $w1$ of the neurons are coded in three columns and four rows, which result from the four independent variables of the IRIS dataset. This time three biases $b1$ need to be added. The code for the hidden layer and the output layer is written accordingly, so that there is again one neuron in the output layer for binary classification.

(Tab. 4) Weights and Biases for 6 Neurons

```
w1 = [[ 0.50914556, -0.18116623, -0.04498423],
      [ 0.33949652, -0.42303845, -0.37400272],
      [-1.4968083 ,  1.2034143 ,  0.95544535],
      [-1.344156  ,  0.39220142,  1.2244085 ]]
b1 = [0.83684736, 0.5311056 , 0.7652087 ]
w2 = [[-2.1645586 ,  1.3892978 ],
      [ 0.43439832, -1.8758974 ],
      [ 0.92036045, -1.5745732 ]]
b2 = [0.9615521, 0.4445824]
w3 = [[ 1.6905344],
      [-2.6346245]]
b3 = [0.4316521]
```

This time, the ReLU activation function was used during pre-training in TensorFlow and Keras for the input layer, followed by the Sigmoid activation function in the hidden layer and in the output layer. This results in the following code in MicroPython (Tab. 5):

(Tab. 5) Neural Network with 6 Neurons

```
yout1 = dense(3, transpose(Xtest), w1, b1, 'relu')
yout2 = dense(2, yout1, w2, b2, 'sigmoid')
ypred = dense(1, yout2, w3, b3, 'sigmoid')
```

This results in an Accuracy of 95%, whereby flexible adjustments are also possible here. The different Accuracies of the two examples demonstrate the importance of the number of neurons and layers as well as the activation functions used. In this case, a simpler neural network seems to be more appropriate.

(V) Summary

This working paper explored the integration of neural networks onto resource-limited microcontrollers and embedded systems like the Raspberry Pi Pico and Raspberry Pi Pico 2, using a TinyML approach. This method enabled the deployment of neural networks directly onto microcontrollers, providing real-time, low-latency, and energy-efficient inference while ensuring data privacy. For this purpose, AI-ANNE: (A) (N)eural (N)et for (E)xploration was introduced, a open source framework that facilitated the transfer of pre-trained models from high-performance platforms like TensorFlow and Keras to microcontrollers, using MicroPython as lightweight and transparent programming languages. The approach demonstrated how key neural network components — such as neurons, layers, density, and activation functions — could be implemented in MicroPython in order to address the computational constraints of microcontrollers and embedded systems. Two neural network examples were presented, either with 8 neurons in 4 layers or with 6 neurons in 3 layers.

Overall, this working paper offered a simple and practical method for deploying neural networks on energy-efficient devices like microcontrollers and how they can be used for practical use or as an educational tool with insights into the underlying technology and programming techniques of deep learning models. AI-ANNE: (A) (N)eural (N)et for (E)xploration is an open source project.

About the Author

- Dennis Klinkhammer completed his doctorate and habilitation at Justus Liebig University Giessen (JLU) and focuses his teaching and research as a social data scientist on the methodological foundations of machine learning and deep learning in the programming languages R and Python. Insights into his teaching and research are available at: https://www.statistical-thinking.de/

Open Source Code

- Both the presented and other examples of AI-ANNE: (A) (N)eural (N)et for (E)xploration are available on GitHub. This includes Jupyter Notebooks for pre-training with TensorFlow and Keras in Python, as well as the parameters to be transferred and the corresponding code in MicroPython. The link to the repository is: https://github.com/statistical-thinking/KI.ENNA/

Sources

- Cioffi, R.; Travaglioni, M.; Piscitelli, G.; Petrillo, A. and F. de Felice (2020): Artificial intelligence and machine learning applications in smart production: Progress trends and directions. In: Sustainability, 12(2), p. 492. https://doi.org/10.3390/su12020492
- Delnevo, G.; Mirri, S.; Prandi, C.; Manzoni, P. (2023): An evaluation methodology to determine the actual limitations of a TinyML-based solution. In: Internet of Things, 22, p. 100729. https://doi.org/10.1016/j.iot.2023.100729
- LeCun, Y.; Bengio, Y.; Hinton, G. (2015): Deep Learning. In: Nature, 521, pp. 436-444. https://doi.org/10.1038/nature14539
- Ray, P. (2022): A review on TinyML: State-of-the-art and prospects. In: Journal of King Saud University - Computer and Information Sciences, 34(4), pp. 1595-1623. https://doi.org/10.1016/j.jksuci.2021.11.019
- Sakr, F.; Berta, R.; Doyle, J.; de Gloria, A and F. Bellotti (2021): Self-Learning Pipeline for Low-Energy Resource-Constrained Devices. In: Energies, 14(20), p 6636. https://doi.org/10.3390/en14206636
- Wulfert, L.; Kühnel, J.; Krupp, L.; Viga, J.; Wiede, Ch.; Gembaczka, P.; Grabmaier, A. (2024): AIfES: A Next-Generation Edge AI Framework. In: IEEE Transactions on Pattern Analysis and Machine Intelligence, 46(6), pp. 4519-4533. https://doi.org/10.1109/TPAMI.2024.3355495

Appendix

(A) Mathematical Basics in MicroPython

These mathematical basics enable matrix multiplication and other important operations for the neural network architecture. The codes (Tab. A) therefore do not need to be adapted in MicroPython!

(Tab. A) Mathematical Basics

```
# Mathematical Basics - I
def zero_dim(x):
    z = [0 for i in range(len(x))]
    return z

# Mathematical Basics - II
def add_dim(x, y):
    z = [x[i] + y[i] for i in range(len(x))]
    return z

# Mathematical Basics - III
def zeros(rows, cols):
    M = []
    while len(M) < rows:
        M.append([])
        while len(M[-1]) < cols:
            M[-1].append(0.0)
    return M

# Mathematical Basics - IV
def transpose(M):
    if not isinstance(M[0], list):
        M = [M]
    rows = len(M)
    cols = len(M[0])
    MT = zeros(cols, rows)
    for i in range(rows):
        for j in range(cols):
            MT[j][i] = M[i][j]
    return MT

# Mathematical Basics - V
def print_matrix(M, decimals=3):
    for row in M:
        print([round(x, decimals) + 0 for x in row])

# Mathematical Basics - VI
def dense(nunit, x, w, b, activation):
    res = []
    for i in range(nunit):
        z = neuron(x, w[i], b[i], activation)
        res.append(z)
    return res
```

(B) Activation Funtions in MicroPython

The following code (Tab. B) demonstrates how the activation functions Sigmoid, ReLU, Leaky ReLU, Tanh and Softmax can be programmed in MicroPython. In TensorFlow and Keras these are already pre-programmed, in MicroPython the programming has to be done manually. Additional activation functions can be added accordingly.

(Tab. B) Activation Functions

```
# Sigmoid
def sigmoid(x):
    z = [1 / (1 + math.exp(-x[val])) for val in range(len(x))]
    return z

# ReLU
def relu(x):
    y = []
    for i in range(len(x)):
        if x[i] >= 0:
            y.append(x[i])
        else:
            y.append(0)
    return y

# Leaky ReLU
def leaky_relu(x, alpha=0.01):
    p = []
    for i in range(len(x)):
        if x[i] > 0:
            p.append(x[i])
        else:
            p.append(alpha * x[i])
    return p

# Tanh
def tanh(x):
    t = [(math.exp(x[val]) - math.exp(-x[val])) / (math.exp(x[val])
      + math.exp(-x[val])) for val in range(len(x))]
    return t

# Softmax
def softmax(x):
    max_x = max(x[val])
    exp_x = [math.exp(val - max_x) for val in range(len(x))]
    sum_exp_x = sum(exp_x)
    s = [j / sum_exp_x for j in exp_x]
    return s
```

(C) Neurons in MicroPython

The predefined activation functions Sigmoid, ReLU, Leaky ReLU, Tanh and Softmax are already pre-programmed for each neuron in MicroPython (Tab. C). Activation functions that are added independently must be added accordingly.

(Tab. C) Neurons

```
# Single Neuron
def neuron(x, w, b, activation):

    tmp = zero_dim(x[0])

    for i in range(len(x)):
        tmp = add_dim(tmp, [(float(w[i]) * float(x[i][j]))
          for j in range(len(x[0]))])

    if activation == "sigmoid":
        yp = sigmoid([tmp[i] + b for i in range(len(tmp))])
    elif activation == "relu":
        yp = relu([tmp[i] + b for i in range(len(tmp))])
    elif activation == "leaky_relu":
        yp = relu([tmp[i] + b for i in range(len(tmp))])
    elif activation == "tanh":
        yp = tanh([tmp[i] + b for i in range(len(tmp))])
    elif activation == "softmax":
        yp = tanh([tmp[i] + b for i in range(len(tmp))])
    else:
        print("Function unknown!")

    return yp
```

(D) German Free Software License

AI-ANNE: (A) (N)eural (N)et for (E)xploration and KI-ENNA: (E)in (N)euronales (N)etz zum (A)usprobieren may be used by anyone in accordance with the terms of the German Free Software License. The German Free Software License (Deutsche Freie Software Lizenz) is a license of open-source nature with the same flavors as the GNU GPL but governed by German law. This makes the license easily acceptable to German authorities. The D-FSL is available in German and in English. Both versions are equally binding and are available at: http://www.d-fsl.org/