BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

OPTIMIZATION OF NEURAL NETWORKS FOR PRINTED ELECTRONICS

OPTIMALIZACE NEURONOVÝCH SÍTÍ PRO TIŠTĚNOU ELEKTRONIKU

PROJECT PRACTICE
PROJEKTOVÁ PRAXE

AUTHOR AUTOR PRÁCE	**NURDAULET TURAR**
SUPERVISOR VEDOUCÍ	**Ing. VOJTĚCH MRÁZEK, Ph.D.**

BRNO 2025

Abstract

Printed electronics have significant operational constraints, making even the simplest full precision neural networks used in typical computers not programmable into them. This work is concerned with feasibility of encoding neural networks (NNs) into printed electronics (PE) through compression techniques.

It evaluates several neural network compression techniques including binarized networks, ternary weight networks, and n-bit quantization. Two complementary experiments were conducted: first, compare compression techniques on well known architectures (LeNet5 and VGG7), and second, using neural architecture search explore optimal architecture compression combinations across the accuracy-complexity trade-off space. Neural architecture search (NAS) was applied to multi-layer perceptron (MLP) and convolutional neural network (CNN) architectures across multiple datasets, generating Pareto-optimal solutions that balance accuracy and computational complexity for printed electronics constraints.

A functioning, configurable, and extendable program for NAS is presented alongside the results of the aforementioned experiments.

Abstrakt

Tištěná elektronika má výrazná provozní omezení, která znemožňují naprogramovat do ní i ty nejjednodušší neuronové sítě s plnou přesností, jaké se používají v běžných počítačích. Tato práce se zabývá proveditelností zakódování neuronových sítí (NN) do tištěné elektroniky (PE) pomocí kompresních technik.

Hodnoceno je několik kompresních technik pro neuronové sítě, včetně binarized NNs, ternary weight networks, a n-bitové kvantizace. Byly provedeny dva doplňující se experimenty: nejprve porovnání kompresních technik na známých architekturách (LeNet5 a VGG7), a poté hledání optimálních kombinací architektury a komprese pomocí neural architecture search (NAS) v prostoru kompromisu mezi přesností a výpočetní složitostí. NAS byl aplikován na architektury multilayer perceptron (MLP) a convolutional neural network (CNN) napříč několika datovými sadami, přičemž byly generovány Pareto-optimální varianty vyvažující přesnost a výpočetní náročnost s ohledem na omezení tištěné elektroniky.

Spolu s výsledky zmíněných experimentů je představen funkční, konfigurovatelný a rozšiřitelný program pro NAS.

Keywords

Neural Network Compression, Printed Electronics, Neural Architecture Search (NAS), NSGA-II

Klíčová slova

Komprese neuronových sítí, Tištěná elektronika, Hledání architektur neuronových sítí

Reference

TURAR, Nurdaulet. *Optimization of neural networks for printed electronics.* Brno, 2025. Project practice. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vojtěch Mrázek, Ph.D.

Contents

List of Figures

Chapter 1

Introduction

Printed electronics have significant operational constraints, with feature sizes typically several microns, and circuits operating at frequencies ranging from a few Hz to a few kHz [3]. Even simple full-precision (FP) convolutional neural networks (CNNs) are unfeasible as-is. This report is concerned with the feasibility of encoding neural networks (NNs) into printed electronics (PE) through compression techniques.

This work evaluates several neural network compression techniques including binarization [5] [8], ternarization [7], and n-bit quantization [5]. Two complementary experiments were conducted: first, comparing compression techniques in established architectures (LeNet5 and VGG7), and second, using neural architecture search (NAS) with Non-dominated Sorting Genetic Algorithm 2 [6] (NSGA2) to explore optimal architecture-compression solutions across the accuracy-complexity trade-off space.

NAS explored architectures for both multilayer perceptrons (MLPs) and convolutional neural networks (CNNs) across multiple datasets, generating Pareto-optimal solutions that balance accuracy and computational complexity for printed electronics constraints.

The source code is available online, enabling further development and collaboration. It is designed with extensibility in mind, allowing additional compression techniques to be incorporated relatively easily. The codebase has a potential to be generalized into a tool with even greater extensibility, facilitating wider adoption. There is also potential for integration with other neural network compression tools, particularly those operating at lower, hardware-oriented abstraction levels, like shown in [1].

Chapter 2

Methodology

To determine the effectiveness of neural network compression techniques, 2 experiments were designed:

1. Compression technique comparison on well-known architectures (LeNet5 and VGG7) to evaluate performance of different compression methods.
2. Architecture optimization using neural architecture search (NAS) to explore NN architecture and compression combinations across the accuracy-complexity trade-off space.

2.1 Evaluation metrics

2.1.1 Accuracy Assessment

Architecture accuracy is measured using stratified k-fold cross-validation with 5 folds. To evaluate an architecture's accuracy, training is repeated multiple times, with the best accuracy result recorded as the architecture's final accuracy score. Early stopping is employed to avoid overfitting.

Stratified K-Fold Cross-Validation

K-Fold Cross-Validation technique splits a dataset into k equal groups (folds), then iteratively uses $k-1$ folds for training while reserving one fold for validation. This process repeats k times, ensuring each fold serves as the test set exactly once. The final performance estimate is the average of all k evaluation scores, providing a more reliable assessment of model generalization capability.

Stratified variation of k-fold cross-validation attempts to include the same amount of each class into each fold. This approach is important for classification tasks with imbalanced datasets, as it prevents individual folds from having disproportionate class distributions that could lead to biased performance estimates.

Early Stopping

Training is halted when validation error begins to increase or plateaus for a specified number of epochs (patience parameter).

2.1.2 Complexity Assessment

Since precise assessment requires implementing or simulating architectures on concrete PE devices - a time and compute-intensive process - a heuristic approximation function was developed. This heuristic estimates computational complexity by counting multiply-accumulate (MAC) operations and applying compression technique specific coefficients. When using ternary compression mode, zero-weight connections would be removed on implementation, which reduces the complexity. This was not considered in the complexity estimation heuristic function. Due to this inaccuracy, a heuristic function has a bias towards undervaluing ternary compression solutions.

No tests were done in order to compare its precision with an actual or synthesized versions of the models.

To see an implementation of the heuristic function, visit appendices at A or the source code provided further in the report.

2.1.3 Compression Techniques

For linear and convolution layers:

- Binarize [5]. During training and inference, parameters are binarized, meaning they are converted to one of $\{-d, d\}$, where d is a scaling factor. (Implementation for convolution layers was taken from the source code of [7].)

- Ternarize [7]. During training and inference, parameters are ternarized, meaning converted to one of $\{-d, 0, d\}$, where d is a scaling factor.

- N-bit quantization. During training and inference, parameters are quantized from $32 \rightarrow$ N bits. Implementation is taken from the source code of [5].

- None. (full-precision, baseline)

For activations:

- Binary activation with Straight Through Estimator (STE) [5]. Inputs are converted to one of $\{-d, d\}$, where d is a scaling factor. Gradients of such activation are estimated by behaving as if there is no compression happening.

- Binary activation with ReSTE. Same as Binary activation, but with different gradient estimation function presented in [8].

- None.

- ReLU. (baseline)

2.1.4 Datasets

Datasets used in this report:

1. Vertebral column [2]

2. Breast cancer [9]

3. Cardiotocography [4]

4. Mini MNIST: A smaller subset of normalized, 28x28 MNIST Dataset. Loaded from pytorch. Chosen because it is used in [5] [7]. Due to upcoming deadlines, only a portion of the dataset was used to shorten the time to run experiments: 4000/70000 images.
5. Mini MNIST 32x32: Mini MNIST Dataset resized to 32x32 pixels. Used for LeNet5 achitecture to have comparible inputs.
6. Mini CIFAR10: A smaller subset of normalized, 32x32 CIFAR10 Dataset. Loaded from pytorch. Chosen because it is used in [5] [8] [7]. Due to upcoming deadlines, only a portion of the dataset was used to shorten the time to run experiments: 8000/60000 images.

2.2 Experiment 1: Evaluate compression on static architectures

Two well-known architectures were selected for evaluation:

1. **LeNet5:** Evaluated on Mini MNIST 32x32 dataset for each combination of compression techniques: 2.1.3.
2. **VGG7:** Evaluated on Mini CIFAR10 dataset for each combination of compression techniques: 2.1.3. Due to starting an experiment late and having an upcoming deadline, this evaluation didn't finish in time and therefore is not included in this report.

In this experiment, in case of usage on N-bit quantization, only a value of N=4 was considered.

2.3 Experiment 2: Run neural architecture search to find optimal architecture and compression modes

Neural Architecture Search (NAS) is an automated technique that designs optimal neural network architectures by evaluating many architectures many times and choosing those that show best properties. Since models under optimization are not very deep, the computational overhead of NAS is manageable, and automating the architecture search saves significant human time that would otherwise be spent on manual experimentation.

This implementation of NAS is based on NSGA2. NSGA2 is an evolutionary algorithm designed for solving multi-objective optimization problems, where multiple conflicting objectives must be optimized simultaneously. The NSGA2 is a suitable algorithm here because we have exactly that problem. We are optimizing for:

- Neural network accuracy
- Neural network complexity

Given that the same data was used for training and validation, these targets are influenced by the NN's architecture. NSGA2 therefore is searching for optimal NN architectures. By comparing different architectures, we address a problem in experiment 1, being that different compression techniques might need different model architectures to perform optimally.

Several NASs were executed for the following datasets: Vertebral Column, Breast Cancer, Cardiotocography, Mini MNIST and Mini CIFAR10 2.1.4.

2.3.1 Search spaces and chosen constants

To search for an optimal architecture, a search space must be defined. Tables 2.1 and 2.2 outline them. NAS parameters and NN architecture properties which were not parametrized and have a constant value used are shown in 2.3

Gene	**Search Space**
Activation function	NONE, BINARY, BINARY_RESTE, RELU
FC Layers Compression Type	BINARY, TERNARY, NBITS, NONE
Input quantization level	1-8
Hidden layers amount	0-3
Per-hidden-layer height	6, 7, 8, 12, 16, 24, 32
If compression type is NBITS, value of N	1-8
Output layer NBITS N value	1-8
Dropout rate	0.0, 0.1, 0.2

Table 2.1: MLP Architecture Search Space

Gene	**Seach Space**
Activation function	NONE, BINARY, BINARY_RESTE, RELU
Conv Layers Compression Type	BINARY, TERNARY, NBITS, NONE
FC Layers Compression Type	BINARY, TERNARY, NBITS, NONE
Conv layers amount	1-4
Per-conv-layer channels	16, 24, 32, 64, 128
Per-conv-layer stride	1-2
Per-conv-layer pooling	1x1 (effectively without pooling), 2x2
Per-conv-layer NBITS N value	1-8
Hidden FC layers amount	0-3
Per-hidden-FC-layer height	6, 7, 8, 12, 16, 24, 32
Per-hidden-FC-layer NBITS N value	1-8
Output layer NBITS N value	1-8
Dropout rate	0.0, 0.1, 0.2

Table 2.2: CNN Architecture Search Space

2.4 Technical Notes

Bias was disabled in convolutional layers since batch normalization layers that are following right after convolutions already contain bias terms, making additional bias parameters redundant.

Hyperparameter	**Value**
Population size	40
Offspring count	8
Generations	5
Mutation Probability	0.9
Mutation Eta	20
Crossover Probability	0.9
Crossover Eta	15
Training epochs	30
Early stopping patience	5
Architecture evaluations	3
Learning rate	0.001
Weight decay	0.00001
ReSTE, o[8]	3
ReSTE, t[8]	1.5

Table 2.3: NAS Hyperparameters

2 servers, one with 4 NVIDIA GTX 1080 GPUs and a second with 4 NVIDIA A5000 GPUs were utilized. Some experiments ran at the same time. A table showing time of execution is at 2.4.

GPU	**Experiment №**	**Description**	**Execution time**	**Batch size**
GTX A5000	1	Evaluation on LeNet5 architecture with MiniMINST32x32 Dataset	11 hours	50
GTX 1080	2	NAS on Cardiotocography Dataset	9 hours	32
GTX 1080	2	NAS on BreastCancer Dataset	2 hours	32
GTX A5000	2	NAS on MiniMNIST Dataset	6.5 hours	50
GTX A5000	2	NAS on Vertebral Dataset	3.5 hours	32
GTX A5000	2	NAS on MiniCIFAR10 Dataset	running	100

Table 2.4: Experiments Runtime

Chapter 3

Results and Discussion

3.1 Experiment 1: Evaluate compression on static architectures

From a plot of the compressed models at 3.1 we can observe that:

- Models that have a ReLU activation function are leading in terms of accuracy.
- Models without activation function follow closely behind ReLU in terms of accuracy, being less complex, but performing a bit worse.
- Models without FC/Conv compression have the highest complexity metric value, meaning they are worst.
- Although the ReSTE gradient estimator is stated be closer to FP gradients than STE[8], this experiment indicates that while complexity is the same, accuracy of models binarized with STE are better than those using ReSTE in every configuration of compression techniques.

From a plot of the pareto front of these models at 3.2, we can observe that:

- 3/4 solutions have ReLU activation function.
- A non-relu solution has a drastically smaller complexity value, with 2.3% accuracy drop compared with its ReLU counterpart.

3.2 Experiment 2: Run neural architecture search to find optimal architecture and compression modes

NSGA2 was executed for only 5 generations - a relatively small number. Therefore, the primary contribution of this section is to show that the entire pipeline is fully operational. We can now conduct more experiments in an unsupervised manner, which can bring dramatic time savings in research of other compression techniques.

The resulting population plots are shown in 3.3, 3.4, 3.5 and 3.6.

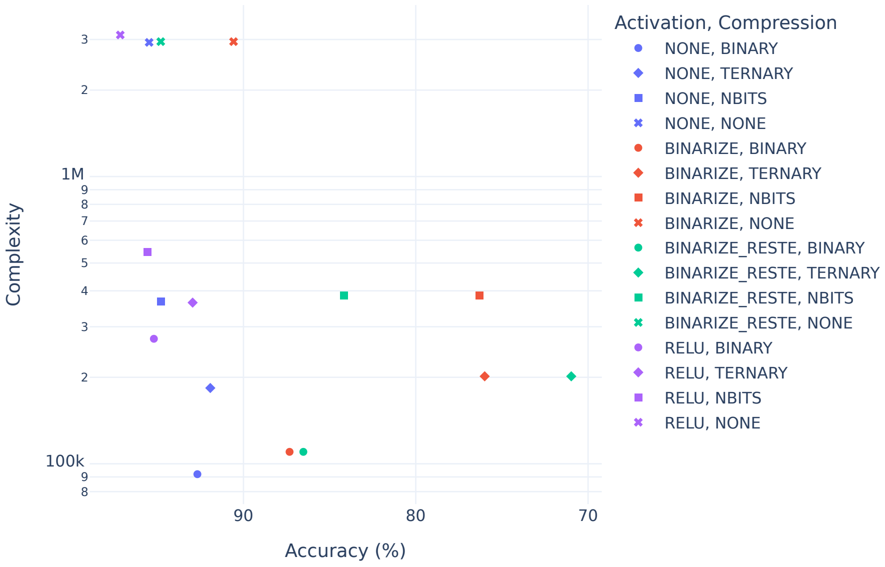

Figure 3.1: Performance of models based on the LeNet5 architecture

3.3 Source Code

Implementation uses Python programming language. Models were constructed using Pytorch. Optimization was run with Pymoo framework, which has NSGA2 algorithm built-in. To make plots, graphs, plotly library was used. An unusually high overall amounts of caffeine were dedicated to finish this report in time.

The source code is available online[1], enabling further development and collaboration. It is designed with extensibility in mind, allowing additional compression techniques to be incorporated relatively easily. The codebase has a potential to be generalized into a tool with even greater extensibility, facilitating wider adoption. There is also potential for integration with other neural network compression tools, particularly those operating at lower, hardware-oriented abstraction levels, like shown in [1].

[1] https://github.com/DiceNameIsMy/vut-ip1-nn-quantization

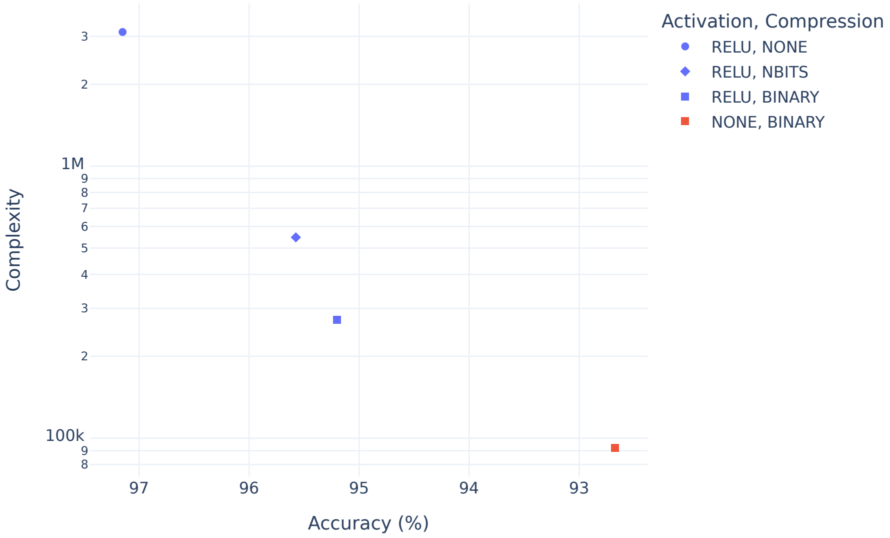

Figure 3.2: Pareto front of models based on the LeNet5 architecture

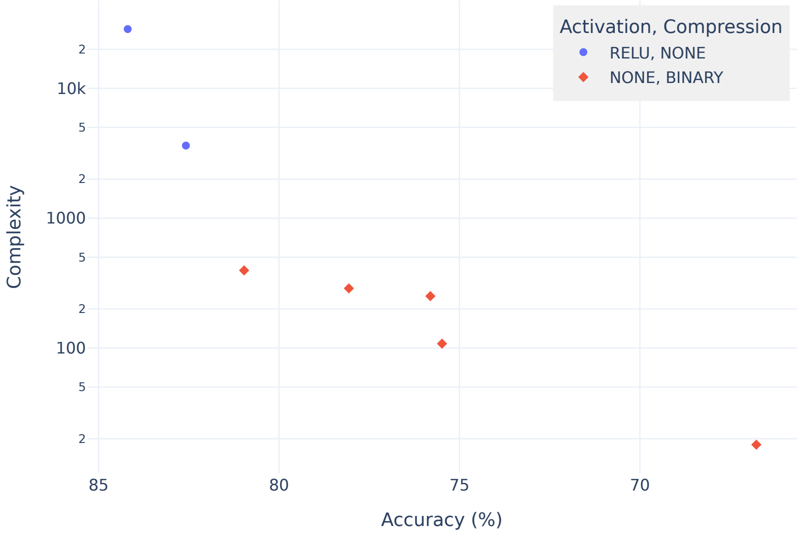

Figure 3.3: Architectures population after running NAS for Vertebral Dataset

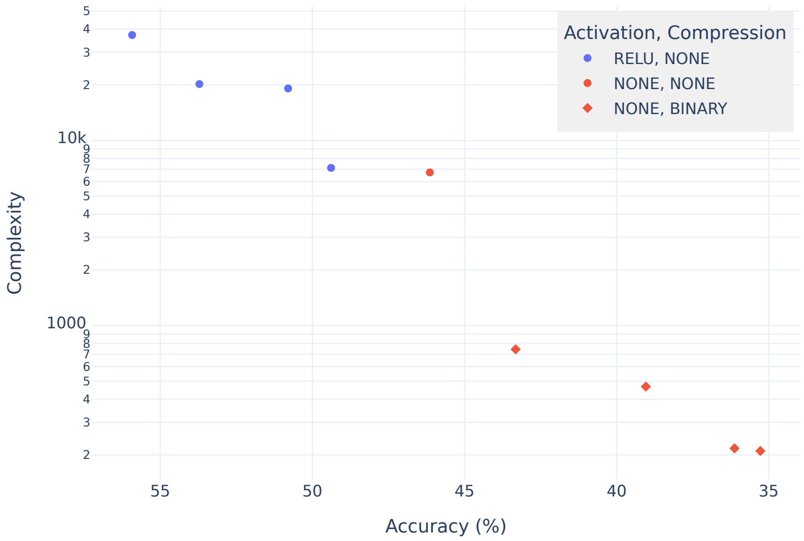

Figure 3.4: Architectures population after running NAS for Cardiotocography Dataset

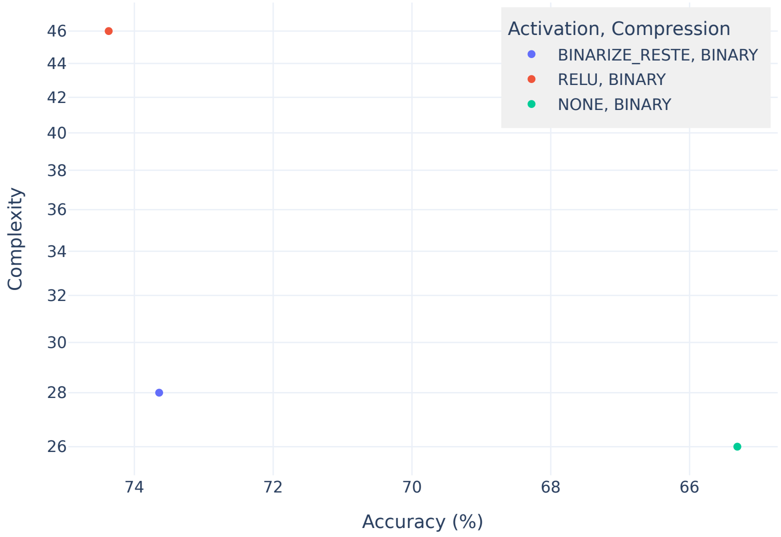

Figure 3.5: Architectures population after running NAS for Breast Cancer Dataset

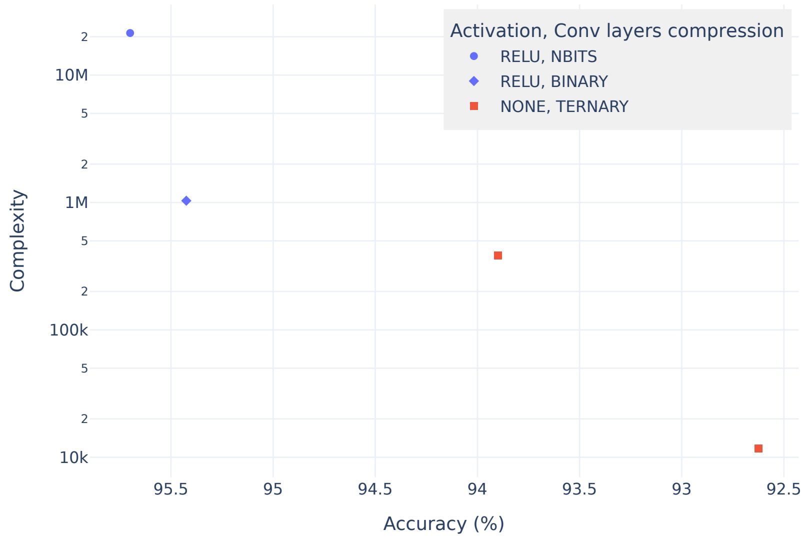

Figure 3.6: Architectures population after running NAS for Mini MNIST Dataset

Chapter 4

Timeline

In this section, a timeline of the work I've made is presented.

4.1 Summer 2024

1. Read a book on Genetic Algorithms[1]
2. Started a couse on ML[2]

4.2 Winter semester 2024/2025

1. Research of NN compression techniques: [5], [8], [7], and many more.
2. Create a Google Colab notebook[3]
3. Implement a configurable MLP. It will be used when running NAS.
4. Implement a heuristic function for estimating complexity of MLPs.
5. Implement NAS for MLP architectures. Pymoo was used as a minimization library. It has an implementation of NSGA2 built in.
6. Implement plotting of the pareto front from the resulting population.

4.3 Summer semester 2024/2025

1. Move the notebook to a github repository and initiate decomposition of the functionality into smaller modules. My notebook became increasingly cumbersome to work with because a) it is a single file and the amount of code was nearing 1000 lines, b) colab was lacking useful shortcuts, syntax highlighting, and refactoring tools, which made editing code time consuming and c) version control was lacking in features.
2. Implement variable-length chromosome encoding to allow per-layer configuration. At the time, chromosome representation was pretty simplistic, with locked amount of

[1] https://www.databazeknih.cz/prehled-knihy/evolucni-hardware-84435

[2] https://mlcourse.ai/book/index.html

[3] https://colab.research.google.com/drive/1aLHvQAZFgA7OtnRm6r2sY7NgP4mGZORw

hidden layers where every layer had the same topology. It is suboptimal, as layers at different depths have different optimum sizes and compression options.

3. Implement a configurable CNN. It was advised to do so; I didn't ask why. My understanding is that MLPs are fairly simple. CNNs are more complex and might be a more interesting, useful, and challenging model type to compress.

4. Implement an even more configurable MLP: per-layer compression, per-layer height, etc.

5. Implement a heuristic function for estimating complexity of CNNs.

6. After NAS has finished, store architecture and weights of best performing population, which is forming a pareto front.

7. Create a CLI to run the training and execution of aforementioned experiments.

Bibliography

[1] Afentaki, F.; Saglam, G.; Kokkinis, A.; Siozios, K.; Zervakis, G. et al. Bespoke Approximation of Multiplication-Accumulation and Activation Targeting Printed Multilayer Perceptrons. In: *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, October 2023. Available at: http://dx.doi.org/10.1109/ICCAD57390.2023.10323613.

[2] Barreto, G. and Neto, A. *Vertebral Column* UCI Machine Learning Repository. 2005. Available at: https://archive.ics.uci.edu/dataset/212/vertebral+column. DOI: https://doi.org/10.24432/C5K89B.

[3] Cadilha Marques, G.; Garlapati, S. K.; Dehm, S.; Dasgupta, S.; Hahn, H. et al. Digital power and performance analysis of inkjet printed ring oscillators based on electrolyte-gated oxide electronics. *Applied Physics Letters*, september 2017, vol. 111, no. 10, p. 102103. ISSN 0003-6951. Available at: https://doi.org/10.1063/1.4991919.

[4] Campos, D. and Bernardes, J. *Cardiotocography* UCI Machine Learning Repository. 2000. Available at: https://archive.ics.uci.edu/dataset/193/cardiotocography. DOI: https://doi.org/10.24432/C51S4N.

[5] Courbariaux, M.; Hubara, I.; Soudry, D.; El Yaniv, R. and Bengio, Y. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. Available at: https://arxiv.org/abs/1602.02830.

[6] Deb, K.; Pratap, A.; Agarwal, S. and Meyarivan, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 2002, vol. 6, no. 2, p. 182–197.

[7] Li, F.; Liu, B.; Wang, X.; Zhang, B. and Yan, J. *Ternary Weight Networks*. 2022. Available at: https://arxiv.org/abs/1605.04711.

[8] Wu, X.-M.; Zheng, D.; Liu, Z. and Zheng, W.-S. *Estimator Meets Equilibrium Perspective: A Rectified Straight Through Estimator for Binary Neural Networks Training*. 2023. Available at: https://arxiv.org/abs/2308.06689.

[9] Zwitter, M. and Soklic, M. *Breast Cancer* UCI Machine Learning Repository. 1988. Available at: https://archive.ics.uci.edu/dataset/14/breast+cancer. DOI: https://doi.org/10.24432/C51P4M.

Appendix A

Heuristic functions

MLP Complexity Heuristic

```
def get_mlp_complexity(self, params: MLPParams) -> float:
        return params.fc.get_complexity()
```

CNN Complexity Heuristic

```
def get_cnn_complexity(self, params: CNNParams) -> float:
        conv_complexity = params.conv.get_conv_complexity()
        fc_complexity = params.fc.get_complexity()
        complexity = conv_complexity + fc_complexity
        return complexity
```

Fully Connected Layers Complexity Heuristic

```
def get_complexity(self) -> float:
        complexity = 0
        prev_layer = self.layers[0]
        without_last_layer = self.layers[1:]
        for layer in without_last_layer:
                mac_ops = prev_layer.height * layer.height
                complexity += (
                    mac_ops
                    * layer.get_compression_complexity_coefficient()
                )
                complexity += (
                    layer.height
                    * self.activation.get_activation_complexity_coefficient()
                )
                prev_layer = layer
        return complexity
```

Convolution Layers Complexity Heuristic

```python
def get_conv_complexity(self) -> float:
        complexity = 0
        in_dimensions = self.in_dimensions
        in_channels = self.in_channels
        for layer in self.layers:
                reduce_image_by = layer.kernel_size // 2 + layer.padding
                out_dimensions = (in_dimensions - reduce_image_by) // layer.stride
                out_channels = layer.channels

                # Convolution complexity
                mac_ops_per_out_channel = in_channels * out_dimensions**2
                mac_ops = mac_ops_per_out_channel * out_channels
                complexity += mac_ops * layer.get_conv_complexity_coefficient()

                # Activation complexity
                out_size = out_dimensions**2 * out_channels
                complexity += (
                    out_size
                    * self.activation.get_activation_complexity_coef()
                )

                # Pooling compexity
                if layer.add_max_pooling():
                        acc_ops = (out_dimensions // 2) ** 2
                        complexity += acc_ops

                in_dimensions = out_dimensions
                in_channels = out_channels
        return complexity

def add_max_pooling(self):
        return self.pooling_kernel_size > 1
```