IEEE Open Journal of the
Industrial Electronics Society

Received 20 December 2023; revised 29 January 2024; accepted 2 February 2024. Date of publication 7 February 2024;
date of current version 22 February 2024. The review of this article was arranged by Associate Editor Juan J. Rodriguez-Andina.

Digital Object Identifier 10.1109/OJIES.2024.3363500

Automatic Synthesis of Recurrent Neurons for Imitation Learning From CNC Machine Operators

HOA THI NGUYEN [1,2], ROLAND OLSSON [1], AND ØYSTEIN HAUGEN [1] (Member, IEEE)

[1]Faculty of Computer Sciences, Engineering and Economics, Høskolen I Østfold Universit, 1757 Halden, Norway
[2]Department of Informatics, University of Oslo, 0373 Oslo, Norway

CORRESPONDING AUTHOR: HOA THI NGUYEN (e-mail: hoan@hiof.no).

This work was supported in part by Arrowhead Tools, ECSEL, under Grant No 826452 (Arrowhead Tools) and in part by the European Union Horizon 2020 Research and Innovation Programme and the member states.

ABSTRACT Analyzing time series data in industrial settings demands domain knowledge and computer science expertise to develop effective algorithms. AutoML approaches aim to automate this process, reducing human bias and improving accuracy and cost-effectiveness. This article applies an evolutionary algorithm to synthesize recurrent neurons optimized for specific datasets. This adds another layer to the AutoML framework, targeting the internal structure of neurons. We developed an imitation learning control system for an industry CNC machine to enhance operators' productivity. We specifically examine two recorded operator actions: adjusting the engagement rates for linear feed rate and spindle velocity. We compare the performance of our evolved neurons with support vector machine and four well-established neural network models commonly used for time series data: simple recurrent neural networks, long-short-term-memory, independently recurrent neural networks, and transformers. The results demonstrate that the neurons evolved via the evolutionary approach exhibit lower syntactic complexity than LSTMs and achieve lower error rates than other networks. They yield error rates 270% lower for the first operation action, while the error rates are 20% lower for the second action. We also show that our evolutionary algorithm is capable of creating skip-connections and gating mechanisms adapted to the specific characteristics of our dataset.

INDEX TERMS CNC machine, evolutionary algorithm, imitation learning, recurrent neural networks, smart manufacturing, time series analysis.

I. INTRODUCTION

Time series analysis plays a crucial role in industrial applications, providing a powerful tool for understanding and optimizing processes. Techniques such as trend analysis and anomaly detection help identify patterns, enabling informed decision-making. In the finance sector, these techniques are applied for market analysis and forecasting. The manufacturing sector employs time series analysis for detecting faults, prolonging the lifespan of equipment, and improving safety measures. Furthermore, with the integration of advanced sensors and Internet of Things (IoT) technologies, various industries are increasingly adopting time series analysis to boost operational efficiency and productivity.

Designing AI algorithms for time series applications is complex and requires both domain knowledge and expertise in machine learning (ML) techniques. An established practice of ML in industries is CRoss Industry Standard Process for Data Mining (IoT), in which an ML project goes through six iterative stages: 1) Business Understanding, 2) Data understanding, 3) Data preparation, 4) Modeling, 5) Evaluation, and 6) Deployment [1].

The process of creating a machine learning model, which happens in stages (2), (3), (4), (5), is primarily a task for ML experts. They spend time examining different models, weighing their advantages and disadvantages. It is important for ML specialists to collaborate with domain experts to fully understand the context of the collected data and the project's overall objectives, navigating the search for the most appropriate model.

The limited understanding of the intrinsic characteristics of a dataset can often introduce biases in steps like feature

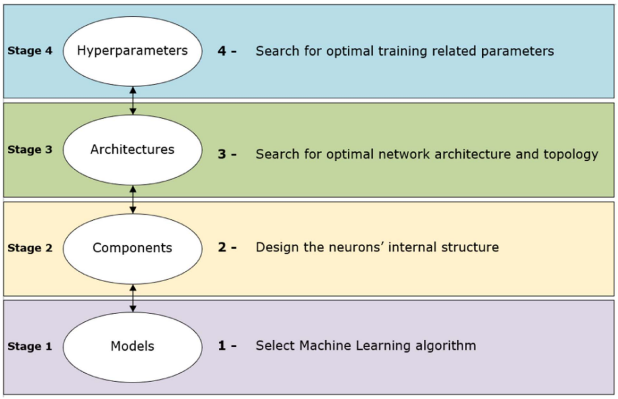


FIGURE 1. Four stages in model generation.

extraction and model design choice [2]. Automated machine learning (AutoML) seeks to reduce these biases by automating many steps in the machine learning pipeline, thus leading to cost reduction and performance improvement [3]. Deep learning techniques have emerged as a powerful approach for time series analysis, as they reduce the need for manual feature engineering. This advantage is achieved by leveraging the deep architectures of neural networks to learn meaningful data representations autonomously [4], [5].

Traditional ML methods encompass stages 1 and 4 in Fig. 1, covering model and hyperparameter search. AutoML techniques for these stages include grid, manual, and random search [6] to determine the optimal set of hyperparameters. The emergence of deep learning methods introduces an additional level in AutoML, denoted as stage 3: architecture search, also known as neural architecture search (NAS). This phase concentrates on identifying the most effective network topology or architecture. The search is performed on a predefined set of components [7], such as convolutional neurons and recurrent neurons, to determine the best arrangement within and between layers.

This article applies an AutoML method that introduces an extra layer to the AutoML hierarchy. It focuses on exploring different internal architectures of the components, specifically targeting recurrent neurons, whereas AutoML traditionally optimizes the overall architecture using fixed and predetermined components.

To address the challenge of crafting the internal structure of a recurrent neuron, we propose using recurrent neurons generated automatically using automatic design of algorithms through evolution (ADATE) [8]. These specialized ADATE-generated recurrent neurons (ARNs) are tailored for each unique dataset. The effectiveness of ARNs has been demonstrated in various applications across different domains, including the prediction of oil well events using the 3 W dataset [9].

This article presents a specific application of ARNs in synthesizing operators' experiences in controlling a computer numerical control (CNC) machine. We collected the dataset over a one-month period from an industrial CNC machine, a Mazak i500 series, which is a state-of-the-art 5-axis CNC machine that is equipped with integrated sensors. The data were collected via MTConnect,[1] with sensor data being streamed alongside operators' action logs. Our objective is to apply imitation learning techniques to capture the operators' skilled reactions and incorporate them into an algorithm. The primary purpose of this algorithm is to reduce the need for continuous monitoring and support operators in decision-making processes. We aim to optimize the operation of the CNC machine and enhance overall productivity. We approached the problem by formulating it as a behavioral cloning task, where we used the same information available to operators during production as input and used their actions as the ground truth. Our objective was to replicate two key actions: controlling the engagement rates for linear feed rate and spindle velocity. We conducted experiments involving the evolution of ARNs using our CNC dataset.

To assess the performance of ARNs, we compared it with other popular neural network algorithms for time series: simple RNNs, LSTMs, independently RNNs (IndRNNs) [10], transformers [11] and one machine learning algorithm that does not use neural networks, for instance, support vector machines (SVMs). These techniques represent the current state-of-the-art in deep learning for time series analysis. Our study showed three key findings. First, the ARNs designed for our dataset outperformed other types of RNNs. It yields error rates 2.7 times lower than the next best-performing network for the first output action and 20% error rates lower than the next best-performing networks for the second output action. The ARNs achieved these superior results while maintaining lower syntactic complexity compared to LSTMs. Second, we provide evidence of the optimality of the ARNs for our dataset by demonstrating that they align well with its characteristics. This suggests that the ARNs have evolved to effectively capture and model the unique patterns and dynamics of our specific data. Finally, our analysis of the ARNs' structures showcases the ability of ARNs to evolve novel architectures featuring gated mechanisms and skip connections through time.

The rest of this article is organized as follows: Section II provides background information, covering different structures of recurrent neurons used in sequential modeling (see Section II-A), automatic design of algorithms through evolution (ADATE) for recurrent neurons (see Section II-B), and related work in CNC production (see Section II-C). In Section III, we introduce the research scenario that serves as the basis for the experimental analysis conducted in Section IV. The obtained results are discussed in Section V, followed by the concluding remarks in Section VI.

II. BACKGROUND

A. SEQUENTIAL MODELING

In this article, we adopt the convention of using bold text to represent matrices and vectors. Superscripts $^{(t)}$ are used to

[1] [Online]. Available: https://www.mtconnect.org/

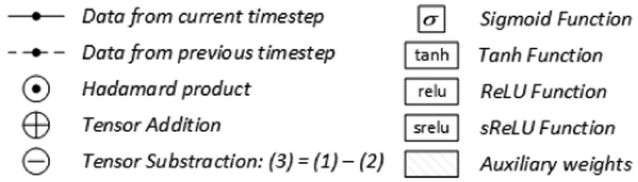


FIGURE 2. Legends for circuit diagrams.

denote the timestep index. For a comprehensive understanding of the symbols and notations used throughout the article, please refer to Fig. 2.

1) RECURRENT NEURAL NETWORKS (RNNS)

RNNs are specifically designed to handle sequential data where information from previous timesteps influences the network's current output. In an RNN cell, the hidden state $h^{(t)}$ is computed at each timestep using the current input token $x^{(t)}$ and the previous hidden state $h^{(t-1)}$. The hidden state serves as the internal memory component within the simple RNN cell [12]

$$\begin{aligned} h^{(t)} &= \sigma(Wx^{(t)} + Rh^{(t-1)} + b_h) \\ y^{(t)} &= f(Uh^{(t)} + b_y). \end{aligned} \tag{1}$$

The output at a given timestep may rely not only on the most recent memory but also on information from many preceding timesteps, indicating a **long-term dependence**. However, unfolding RNNs along the temporal sequence generates a very deep computational graph. When optimizing such networks using backpropagation, the propagated error signal tends to diminish with each timestep, causing the vanishing gradient problem [2], [13].

We will refer to this architecture as the "simple RNN" architecture to differentiate it from other variations.

2) LONG SHORT-TERM MEMORY

In 1997, Hochreiter and Schmidhuber introduced long short-term memory (LSTM) to address the limitations of traditional RNNs. LSTM incorporates a gated mechanism to regulate the flow of information, employing three gate blocks: the forget gate $f^{(t)}$, the input gate $i^{(t)}$, and the output gate $o^{(t)}$ [14]. Over the past two decades, LSTM has proven to be one of the most successful RNN structures, significantly contributing to fields like speech recognition and natural language processing.

This study employs a popular variant of LSTM known as LSTM with peephole connections [15]. To mitigate the vanishing gradient problem, an ideal activation function should sustain the gradient for long-term dependencies. In LSTM, the *sigmoid* function σ is utilized as gates to regulate the flow of information, while the *tanh* function (h) is used to squash the output. In line with recommendations from previous studies, an additional bias is introduced to the forget gate [16]. The circuit diagram for LSTM is shown in Fig. 3. Its mathematical

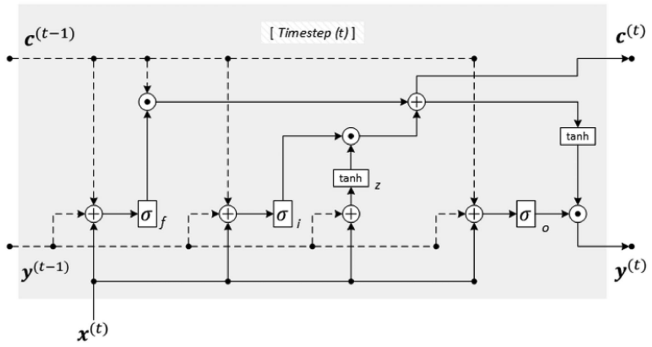


FIGURE 3. Circuit diagram for LSTM.

formulas are as follows:

Input block:

$$z^{(t)} = W_z x^{(t)} + R_z y^{(t-1)} + b_z$$

Forget gate:

$$f^{(t)} = \sigma(W_f x^{(t)} + R_f y^{(t-1)} + p_f \odot c^{(t-1)} + b_f + 1)$$

Input gate:

$$i^{(t)} = \sigma(W_i x^{(t)} + R_i y^{(t-1)} + p_i \odot c^{(t-1)} + b_i)$$

Output gate:

$$o^{(t)} = \sigma(W_o x^{(t)} + R_o y^{(t-1)} + p_o \odot c^{(t-1)} + b_o)$$

Cell memory:

$$c^{(t)} = z^{(t)} \odot i^{(t)} + c^{(t-1)} \odot f^{(t)}$$

Output

$$y^{(t)} = h(c^{(t)}) \odot o^{(t)}.$$

The values of gates f, i, and o in LSTM are dynamically adjusted based on the context provided by input information, previous recurrent states, and memory states. Through the mechanisms of memory retention and information "forgetting," the LSTM cell can, at each timestep, selectively choose and preserve long-term dependencies. This dynamic selection process leads to variable-sized long-term dependencies at each timestep.

However, LSTM has faced criticism for its complexity [16]. The effects of different gates vary across datasets. Jozefowicz et al. [16] reported a significant drop in performance upon removing the forget gate, while the output gate can be removed with minimal impact. On the other hand, Greff et al. [15] reported contrasting results, noting that removing either the output gate or the forget gate had a significant adverse effect on the performance of their datasets. These divergent observations emphasize the dataset-dependent nature of gate functionality in LSTM networks. Variants of LSTM, such as gated recurrent unit (GRU) networks, have been developed as simpler alternatives. GRU, equivalent to LSTM with $f^{(t)} = 1 - i^{(t)}$ [15], [17], does not possess an explicit memory cell

and exposes the entire hidden state without output gate control. Despite its simplicity, GRU remains highly competitive with LSTM and even outperforms it in many tasks [18].

3) ATTENTION MODELS AND TRANSFORMERS

The sequential nature of RNNs is not without its drawbacks, such as the nearby context bias, where information from recent timesteps has a greater influence on the current timestep compared to much earlier timesteps. To address this limitation, the attention mechanism was introduced by Bahdanau et al. [19] for neural machine translation. This mechanism enables the creation of a context summarization that aligns the input sequence with the output sequence.

The transformer models, introduced by Vaswani et al. [20], employed a self-attention mechanism, where the input sequence attends to itself, identifying relevant relationships between components within the same time series sequence. Unlike RNNs, the attention mechanism does not inherently consider the sequential nature of time series. As a result, additional input of positional encoding is required to inform the algorithm about the sequential order of the time series.

Originally developed for neural machine translation, the transformer architecture has gained popularity in time series analysis. The clear advantage of transformers over RNNs lies in their global associative memory. However, the effectiveness of transformers for time series analysis remains a subject of debate [21]. Nevertheless, researchers have explored various adaptations of transformers models tailored for time series analysis. Several of these variations focus on tasks such as long sequence time series forecasting or univariate forecasting [22], [23]. In this study, we chose a version of the transformer models developed by Zerveas et al. [11] specifically designed for multivariable time series analysis that fits our purpose. This variation removes the decoder part of Vaswani's model and replaces the deterministic sinusoidal encodings with fully learnable positional encodings.

4) OTHER DESIGNS

Mikolov et al. [24] introduced a differentiable context layer, denoted as s_t, in conjunction with the simple recurrent layer. This context layer aims to slow down the changes occurring in the hidden layer, thereby preserving long-term dependencies within the model. Another approach to enforcing long-term dependence is through time skipping, which avoids the need for increasing the number of gate parameters. Campos et al. [25] used a binary update gate to determine whether a state should be updated or copied from the previous state. Inspired by skip connections in CNNs, Chang et al. [26] proposed a dilated RNN structure that incorporates skip connections through time.

In the realm of language models, external memory systems such as the neural cache have been employed [27], drawing inspiration from cache memory in computer systems. The neural turing machine, developed by Graves et al. [28], introduced an external memory component while utilizing LSTM as a controller for reading from and writing to the memory.

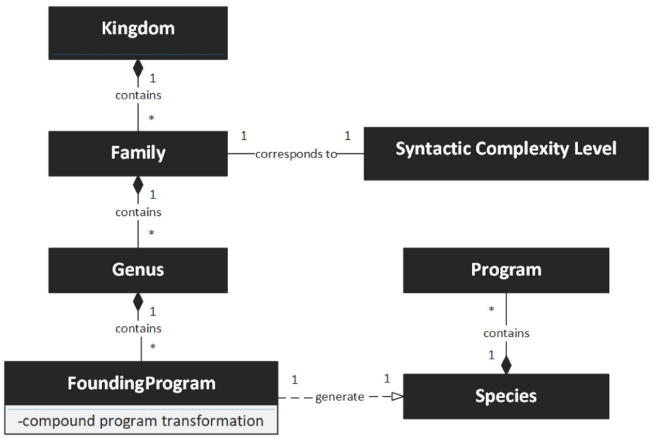


FIGURE 4. Structure of a kingdom of programs in ADATE.

These various techniques and architectures showcase the wide range of approaches taken to address the challenges of preserving long-term dependencies and enhancing the memory capabilities of RNN. In this study, we chose the independently recurrent neural network (IndRNN) developed by Li et al. [10] to represent other variations of RNNs. In IndRNN, neurons within a layer operate independently from one another (2). By stacking multiple layers of IndRNN (3), the architecture enables effective training of deeper and longer IndRNN networks

$$h^{(t)} = \sigma(Wx^{(t)} + u \odot h^{(t-1)} + b) \tag{2}$$

$$h_n^{(t)} = \sigma(w_n x^{(t)} + u_n h_n^{(t-1)} + b_n). \tag{3}$$

B. AUTOMATIC DESIGN ALGORITHMS THROUGH EVOLUTION

Evolutionary computing is rooted in the principles of natural evolution and selection. The fundamental concept involves creating a set of possible solutions for a specific problem. Over successive generations, these solutions undergo evolution transformations, aiming to improve the most effective solutions as they progress.

1) *Population Structure:* ADATE employs a hierarchically structured population called a kingdom as shown in Fig. 4. A kingdom is comprised of multiple families, each corresponding to a level of syntactic complexity. One family consists of many genera. Each genus has a group of founding programs. One founding program generates a species of many programs. Programs within a single species share similarities and often occupy the same level or plateau in the fitness landscape [29].
2) *Program Transformations:* The primary transformation technique is replacement (R), which comes in three forms: replacing an entire small subexpression, reusing a complete subexpression, and reusing parts of a subexpression. The replacements that preserve semantics and

do not deteriorate the program's performance are a special subset of R known as replacement preserving equality (REQ). REQs are expensive to find and can be seen as a neutral walk within the search landscape. The neutral walks play a crucial role in both machine search algorithms and natural evolution. They allow the exploration of a plateau in the search landscape and the identification of a transition point to the next level. Additionally, ADATE incorporates abstraction (ABSTR) for generating new functions and case distribution (CASE-DIST) for adjusting the scope of variables and functions. Both ABSTR and CASE-DIST transformations are considered neutral and pose fewer combinatorial challenges compared to R and REQ transformations [29].

3) Recurrent neuron representation: in ADATE, recurrent neurons are represented as functional programs written in standard ML (SML) [30]. An example of how an LSTM neuron is written in SML can be found in Appendix A. The novelties of ARNs are attributed to three main building blocks:

a) Four memory states per neuron instead of one in a typical recurrent neuron as in Section II-A

$$f(x_t, s_{0,t}, s_{1,t}, s_{2,t}, s_{3,t}, y_t) = (s_{0,t+1}, s_{1,t+1}, s_{2,t+1}, s_{3,t+1}, y_{t+1})$$

where f denotes the neuron.

b) Innovative activation functions can be created through an evolutionary process using a predefined set of activation functions, namely $tanh$, $relu$, and $srelu$. The $srelu$ is a leaky linear approximation of $tanh$; see (4).

c) The recurrent output and memory states can be in the form of *self* or *others*.

During the evolutionary process, various combinations of these building blocks are used to create individual programs. For example, by nesting the $srelu$ functions, the evolutionary process can roughly adjust the slope of $srelu$ [(4), (5)]

$$srelu(x) = \begin{cases} -1 + 10^{-2} \cdot (x+1), & x < -1 \\ x, & -1 \le x \le 1 \\ 1 + 10^{-2} \cdot (x-1), & x > 1 \end{cases} \tag{4}$$

$$srelu(srelu(x)) = \begin{cases} -1 + 10^{-4} \cdot (x+1), & x < -1 \\ x, & -1 \le x \le 1 \\ 1 + 10^{-4} \cdot (x-1), & x > 1. \end{cases} \tag{5}$$

To accommodate the evaluation of these programs, the following weight mappings are defined:

a) Five weight mappings for input x_t: $W_{(x,0)}$, $W_{(x,1)}$, $W_{(x,2)}$, $W_{(x,3)}$, $W_{(x,4)}$

b) Five weight mappings for "other" memory states that are hollow weight matrices with zeros along their diagonal to exclude "self": $U_{(s,0)}$, $U_{(s,1)}$, $U_{(s,2)}$, $U_{(s,3)}$, $U_{(s,4)}$.

c) Five weight mappings for "other" recurrent outputs that also are hollow weight matrices: $U_{(y,0)}$, $U_{(y,1)}$, $U_{(y,2)}$, $U_{(y,3)}$, $U_{(y,4)}$.

d) Auxiliary weights a_i are multiplied elementwise (Hadamard product). The number of auxiliary weights varies depending on the dataset.

The total runtime of ADATE is linked to the average time it takes to execute each newly created program, multiplied by the total number of programs executed. A cost limit determines the number of offspring for a program. The cost limit is increased as ADATE progresses. However, the kingdom cardinality and the cost limit are proportional to the maximum program size. New and better programs frequently replace some in the kingdom, preventing an explosion of the number of programs.

C. RELATED WORKS IN COMPUTER NUMERICAL CONTROL PRODUCTION

CNC production consists of two primary stages: the engineering and operational phases (see Fig. 5). The engineering phase operates within the realm of information, where engineers create blueprints for machine components. This phase's research focuses on overcoming challenges related to optimizing toolpaths in terms of energy efficiency, speed, and precision ([31], [32], [33], [34]). Once the blueprints are finalized, they are transmitted to the CNC machines for actual production. CNC machine operators are tasked with loading materials, overseeing the production process, and conducting quality checks. Research in this phase primarily addresses machine monitoring, encompassing aspects like tool condition monitoring, tool wear monitoring, and quality control ([35], [36], [37]), which fall under the domain of CNC operators. While sensors are optional in the information phase [34], they play a crucial role in this realm, dealing with tangible properties not always represented in the information sphere, such as vibration and temperature. This is why digital twins have emerged as a valuable technology, offering visualization and monitoring capabilities otherwise unavailable. However, the effectiveness of sensors heavily depends on their type and installation location [38]. Dealing with real-world properties also leads to a significantly larger parameter space. Consequently, while digital twins hold promise for machine monitoring, their development is still in its infancy, facing challenges in creating a highly realistic virtual representation of the CNC machine [39]. On the other hand, the advent of IoT technologies provides an opportunity to leverage machine learning in addressing these issues, owing to the massive amount of data they generate [38]. The use of real-time data for operational purposes has been explored in previous studies. Moreira et al. [37] developed a real-time monitoring and control system incorporating a model for predicting surface roughness and a neuro-fuzzy inference system. Their experiment demonstrated that their system could achieve superior surface quality compared to human operators. Sakarinto et al. [40] proposed a decision support system for sharing knowledge and expertise among operators. The foundation of their expert system

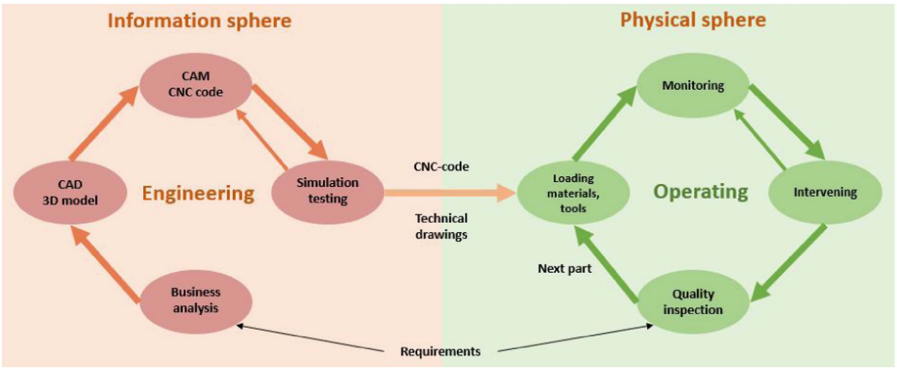


FIGURE 5. **Production procedure of a CNC machine.**

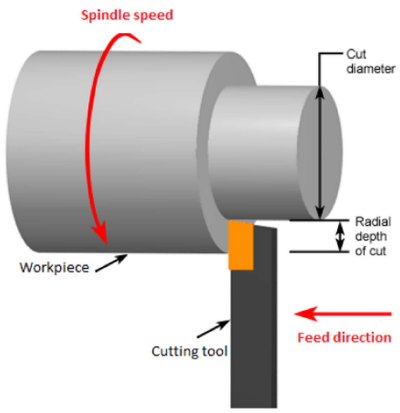
lies in a manually curated knowledge base. Both approaches required expert familiarity with CNC machinery and did not fully utilize Big Data resources.

III. RESEARCH SCENARIO

Mekanisk Service Halden,[2] our industrial partner, is a provider of mechanical services specializing in the precise machining of parts. They possess several cutting-edge CNC machines from the Mazak i-series that are equipped with sensors and offer real-time information services through MT-Connect. The company is interested in exploring potential business opportunities, including optimizing human resources for enhanced productivity. While existing research has predominantly focused on data-driven applications for CNC machines, such as tool condition monitoring and energy consumption optimization ([31], [32], [34]), there has been a notable gap in the investigation of methods to improve human productivity in CNC production. Our research aims to synthesize operators' experiences into AI algorithms. The primary goal of our project is to relieve the burden of constant monitoring for experienced operators while providing training support for novice operators.

Fig. 6(a) shows the control interface of a Mazak machine. The yellow box indicates the current commands of the numerical control (NC) program. While the linear feed rate and spindle velocity are initially programmed into the NC program [see Fig. 6(b)], operators still have the ability to adjust the engagement rates (%) of these settings using the knobs in the blue box. When set at 100%, the Mazak machine operates at the full speed specified in the NC program, while a value of 0% brings the machine's speed to a complete halt. Numerous factors, such as tool condition, may necessitate operator intervention to modify these rates. Operators rely on the information provided on the control interface to make informed adjustments. The Mazak machine does not accept direct control inputs from another computer. Therefore, an operator is required to execute control tasks via the control interface. The AI algorithms serve as a decision support system,

(a)



(b)

FIGURE 6. **(a) Control interface for Mazak Integrex i500. The yellow box shows the current commands. The green box displays cutter positions. The pink box shows cutter loads. The blue box contains knobs to adjust engagement rates. (b) Turning spindle speed and feed rate. Figure adapted by the authors from [41].**

[2][Online]. Available: https://www.mekservice.no/

TABLE 1. List of Input Sensor Signals

Category	Signal	Unit	Description
Axes - Positions	X, Δ	mm	Linear positions of the cutting tools
	Y, Δ	mm	
	Z, Δ	mm	
	B	degree	B angle of the cutting tools
	C	degree	C angle of the workpiece
Axes - Feed rates	X_frt, Δ	mm/s	Linear feed rate of the cutting tools
	Y_frt, Δ	mm/s	
	Z_frt, Δ	mm/s	
	MS_rpm, Δ	rpm	Milling spindle's rotary velocity
	TS_rpm, Δ	rpm	Turning spindle's rotary velocity
Axes - Loads	X_load, Δ	%	Linear load on the cutting tools
	Y_load, Δ	%	
	Z_load, Δ	%	
	MS_load, Δ	%	Milling spindle's load
	TS_load, Δ	%	Turning spindle's load
Axes - Temperature	X temp	celsius	Temperature of X servo motor
	Y temp	celsius	Temperature of Y servo motor
	Z temp	celsius	Temperature of Z servo motor
	MS temp	celsius	Temperature of milling spindle motor
	TS temp	celsius	Temperature of turning spindle motor

[s, Δ] *Signal* s_t, *Rate of change* $\Delta s_t = s_t - s_{t-1}$

TABLE 2. List of Output Signals

Category	Signal	Unit	Description
Controller	**Fovr**	%	Feed rate Override - engagement rate of linear feed rate
	Sovr	%	Spindle speed Override - engagement rate of rotary velocity

allowing a single operator to monitor multiple CNC machines simultaneously, boosting overall productivity.

1) The engagement rate of linear feed rate controls how much feed rate is allowed for the movement of the tools along (x, y, z) (*Fovr* - feed rate override).
2) The engagement rate of spindle velocity controls the rotary velocity allowed for the milling and turning process (*Sovr* - Spindle velocity override).

Their expertise is refined and developed over years of hands-on experience, making training new operators costly. Given this scenario, imitation learning emerges as a viable approach due to the availability of extensive historical logs documenting operators' actions. We hypothesize that by providing machine algorithms with the same information as operators during production, we can train them to synthesize the operators' experiences. Behavioral cloning is a simple, straightforward imitation learning technique for this task, where the goal is to establish a mapping between inputs and the actions of expert operators. The objective function is $f : X_t \rightarrow y_{t+g}$ in which:

1) $X_t \in \mathbb{R}^{N \times M}$ is a vector $X_t = (x_t, x_{t-1}, \ldots, x_{t-n})^T$, x_t is a vector with M features at timestep t. In this case, M = 33 include 20 feature signals and 13 rates of changes $\Delta s_t = s_t - s_{t-1}$ (see Table 1). The feature signals contain information regarding movements, temperatures, and loads on the cutting tools. They match the information on the control interface [see Fig. 6(a)]. N is the window size.
2) $y_t \in \mathbb{R}^2$ is a vector $y_t = (Fovr_t, Sovr_t)$; g is the time into the future that the model forecast. The AI models predict operators' decisions after the g time gap. This is to prevent shortcut learning and to accommodate the decision support system. g is set to one second in the experiment.

Our hypothesis assumes two main characteristics of our dataset:

1) Our observation indicates that operators consistently rely on the most recent sensor measurements to make decisions. This aligns with the characteristics of human psychology, as the human short-term memory capacity is generally limited to processing and retaining around 7 to 10 chunks of information [42]. Consequently, the dataset exhibits a fixed-sized long-term time series dependence. This contrasts the variable-length dependencies often encountered in other time series problems, such as natural language or audio processing.
2) Operators make distinct decisions on *Fovr* and *Sovr* using the same source of information. The extent to which these two output signals, *Fovr* and *Sovr*, might influence each other is unknown. However, preliminary tests revealed that creating a neural network that simultaneously calculates both output signals resulted in poorer performance than developing separate machine learning models for each output signal.

IV. EXPERIMENT

A. DATASET PREPARATION

The data were gathered over a month from the Mazak Integrex i500. During this time, the CNC machine repeatedly manufactured identical units of a product. The maximum data frequency is 4 Hz. A sensor's data are only updated if it

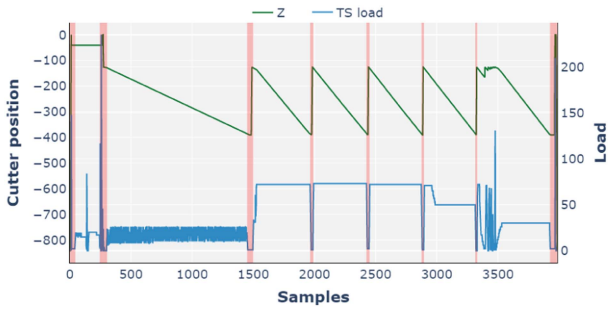


FIGURE 7. Green line visualizes the toolpath on *Z*-axis. Transient periods are highlighted as light pink when the cutter quickly moves to the next position for the cutting path.

changes. This sometimes results in irregular data frequency. The raw data were interpolated to the maximum of 4 Hz to make it consistent.

We segmented the sensor data into distinct episodes, each representing the production of a single machine part. The *Chuck state* signal was used to identify the start and end of one episode. A chuck is a device that secures and rotates the workpiece during machining time. It switches from OPEN or UNLATCH to CLOSED as raw material is loaded into the CNC machine, and vice versa when it is done. The machine operates on the part only when the chuck is CLOSED. There were periods when the CNC machine halted, for example, during lunch break. To filter out these inactive intervals, we relied on the *Execution* signal. We removed sequences when the cutting tools moved to the next ready position as they are transient and could increase the variance of our dataset (see Fig. 7).

We limited the scope to just one type of cutting tool to simplify the parameters. We specifically chose cutting tool ID 1 for this purpose, as it was the most commonly used tool and always positioned at the start of each cycle. Data samples were generated using the window-slicing technique. In preliminary tests, we tried three options for window sizes 20, 40, and 80 data points and found out that a window size of 40 data points gives enough information for good prediction without sacrificing the model's performance. The workpiece surface transitioned from rough to smooth throughout the machining process, while the cutting tool gradually wore out. These factors influenced the operators' actions. The samples were kept in chronological order before being divided into training, validation, and testing sets, following 80-10-10 ratios. This approach aimed to ensure that each set contained a proportional distribution of samples from the early stages of machining to the later stages.

The sensor signals were centered and scaled as recommended by LeCun et al. [43] to facilitate neural network training

$$s_{\text{scaled}} = \frac{s - \mu}{\sigma}$$

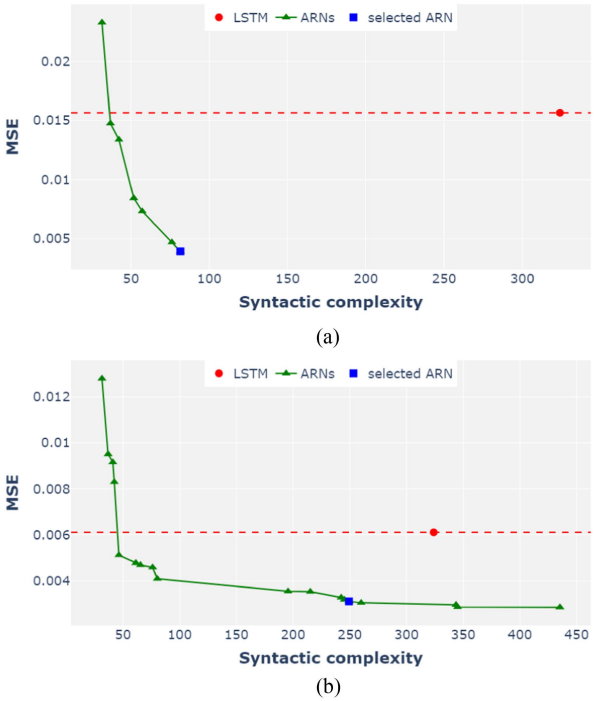


FIGURE 8. Pareto fronts of neurons generated for *Fovr*(a), and for *Sovr* (b).

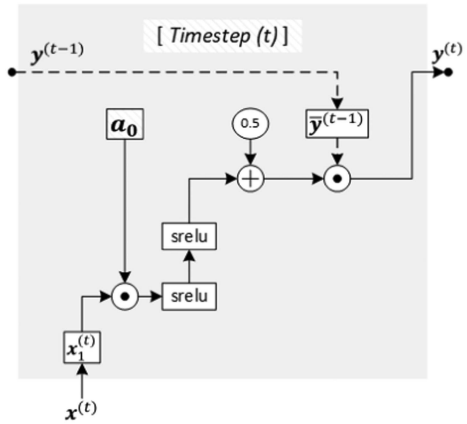


FIGURE 9. Circuit diagram for the selected neuron for *Fovr* (6).

in which μ is mean and σ is standard deviation. Standardization is a common practice in regression problems. It helps the models to learn more quickly and easily. The effects of standardization vary depending on the learning models. In the case of neural networks, standardization positions input into the same scale, thus preventing saturation and speeding up convergence.

For the output signals, namely *Fovr* and *Sovr*, we developed two separate machine learning models. We used mean squared error (MSE) as both the loss function and the metric for evaluation.

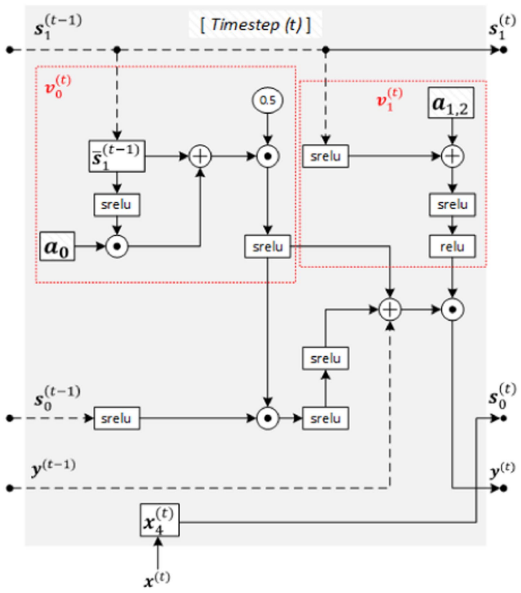


FIGURE 10. Circuit diagram for the selected neuron for *Sovr* (7).

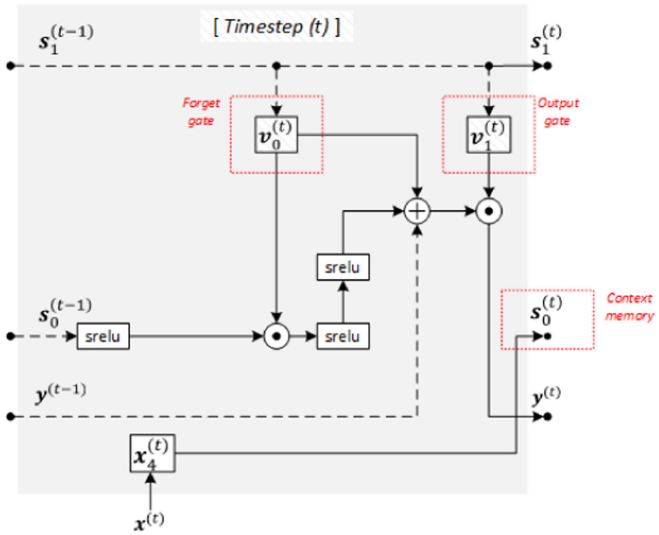


FIGURE 11. Equivalent of the selected neuron for *Sovr* after renaming the constant expressions (7).

B. GENERATING ARNS

1) STEPS

In accordance with the methodology depicted in Fig. 1, the chosen model for this study is RNNs (stage 1). The evolutionary search algorithm employed in this research is tasked with identifying the most effective configuration for the internal structure of these recurrent neurons (stage 2). The network is designed as a single layer of RNN neurons, which is then followed by a feedforward layer (stage 3). The evolutionary algorithm can evolve neurons that adapt to both the characteristics of the dataset and a specific set of hyperparameters. Therefore, in stage 4, hyperparameters can be randomly initiated. In our approach, LSTM networks serve as the benchmark for our evolutionary search process. A thorough hyperparameter optimization was conducted for the LSTM to establish a robust baseline. The detailed steps of this process are outlined below.

1) We adopted the search strategy proposed by Choi et al. [44] for optimizing the hyperparameters for LSTM, incorporating a custom learning schedule

$$\eta^{(t)} = \begin{cases} d^{(t)} \cdot \eta_0, & t \leq T \\ \alpha \eta_0, & t > T \end{cases}$$

$\eta^{(t)}$ is learning rate at time t, η_0 is the initial learning rate, α is decay factor, T is the number of training steps, and $d^{(t)}$ is linear decay function

$$d^{(t)} = 1 - (1 - \alpha)\frac{t}{T}.$$

2) We conducted a search to determine the optimal number of LSTM neurons, considering a predefined set of choices in the range 2^i for $i = 1, 2, \ldots, 7$.
3) The same set of hyperparameters was employed as input for ADATE to evolve ARNs.

Two separate evolutionary processes were carried out for the two output signals *Fovr* and *Sovr*, enabling an examination of how the evolutionary processes treated each signal. The evolution was run on eight servers, where each server had dual AMD EPYC 7551 32-Core CPUs and 256 GB RAM. We used a total of 512 processes running in parallel and communicating using our own TCP/IP library. The time for the first evolution was three days, whereas the time for the second was one week. The servers we used were from 2017. The same experiments could be run equally fast on just one server from 2023 with around 200 cores. In addition, using 32-bit floating point instead of 64-bit can speed up the process.

It is important to note that the mentioned computational resources and time requirements are related to the model creation stage, which is traditionally done manually by ML specialists. Our algorithm is a part of the AutoML stack seeking to automate the process. In terms of training time and resources, there was not a notable difference between using LSTM and ARN-generated neurons. The evolved neurons have lower syntactic complexity than LSTM. Any standard computer is adequate for training and using the model in practice, that is, running it to make inferences. Additionally, the inference speed is more than sufficient to match the real-time data frequency of 4 Hz.

2) PARETO FRONT

A Pareto Front refers to a collection of Pareto efficient solutions, which are regarded as optimal because no other solutions can surpass them without compromising at least one criterion. The concept of a Pareto front is particularly useful when selecting solutions that involve a set of tradeoff constraints [45]. In evolutionary computation, Pareto optimality is commonly employed to identify the most promising candidates in each generation [46].

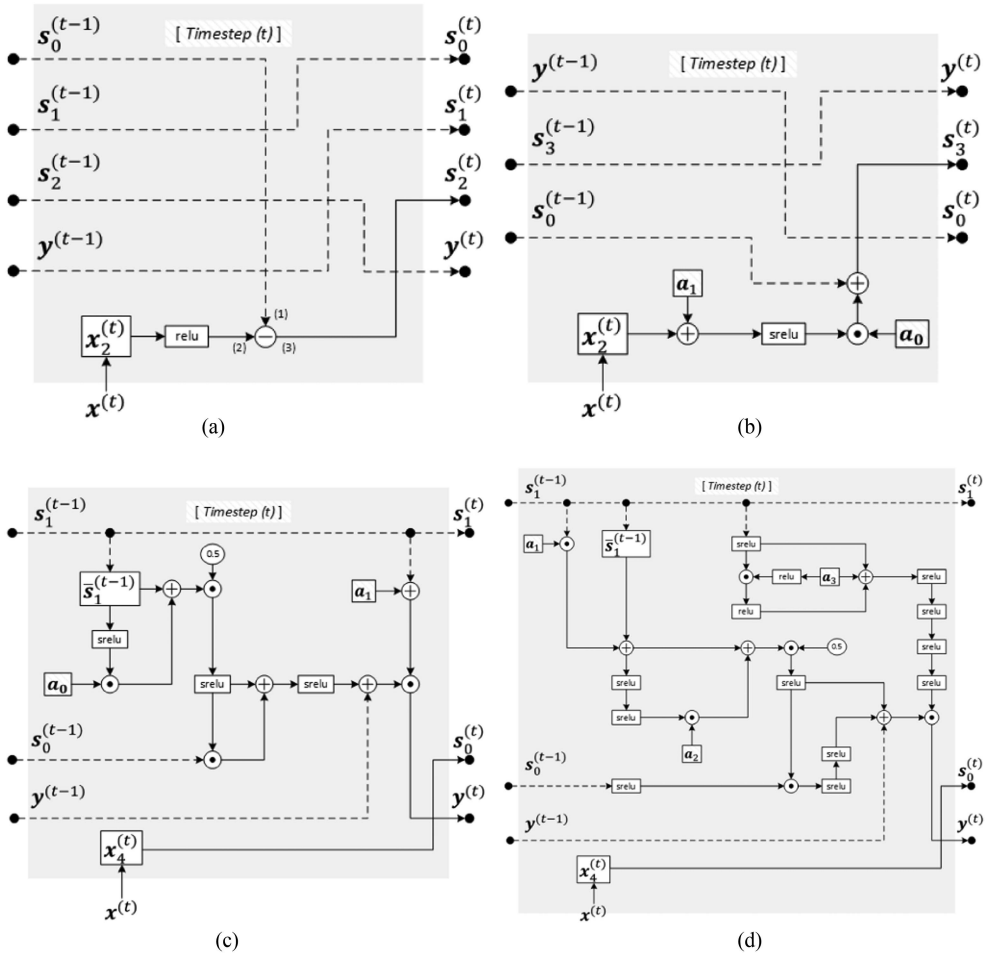


FIGURE 12. Circuit diagrams for neurons generated from Pareto Front for *Sovr*. (a) and (b) are the smallest and biggest neurons in the first cluster, respectively. (c) and (d) are the smallest and biggest neurons in the second cluster.

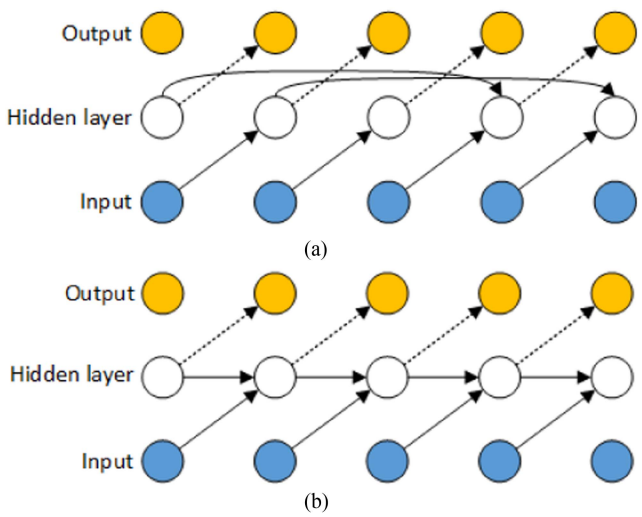


FIGURE 13. Three layers structures for the smallest neuron in the first group (a), and for all neurons in the second group (b).

Regarding ARNs, the selection of neurons is based on two specific constraints: syntactic complexity and the chosen metric for the dataset, in this case, mean squared error (MSE). Syntactic complexity can be understood as the following: all the nodes in the syntax tree of the program will have an occurrence probability. The syntactic complexity of the program can be expressed as the sum of the base-2 logarithms of these occurrence probabilities. Neurons with simpler syntactic complexity are less prone to overfitting.

Fig. 8 shows the Pareto front for the *Fovr* signal and the *Sovr* signal. The LSTM neuron has a complexity of 324 bits. It is expressed as the red dot in Fig. 8. Note that the evolutionary process for *Sovr* lasted for almost a week, while the same process for *Fovr* ended after three days. The search for *Fovr* stopped quicker as *Fovr*'s Pareto Front had already achieved MSEs of 0.0039, four times smaller than for LSTM and with a complexity of only 81.55 bits.

The search for *Sovr* lasted longer as the metric for the dataset was given higher priority among the constraints. The Pareto front for *Sovr* reached a plateau when the complexity

TABLE 3. Median and Standard Deviations of MSE Values Obtained From Eight Runs for *Fovr* (a) and *Sovr* (b). The Best Results are Highlighted in Red, and the Runner-Up Results are in Blue

Model	Val		Test	
	Median	Std	Median	Std
SVM*	0.179	-	0.134	-
SimpleRNN	0.1231	0.0343	0.1400	0.0332
IndRNN	0.1717	0.0413	0.2861	0.0509
LSTM	0.0167	0.0033	**0.0037**	**0.0025**
Transformer	0.0217	0.0023	0.0153	0.0069
ARN	0.0060	0.0012	**0.0015**	**0.0006**

(a)

Model	Val		Test	
	Median	Std	Median	Std
SVM*	0.018	-	0.011	-
SimpleRNN	0.0231	0.0105	0.0249	0.0184
IndRNN	0.1535	0.1315	0.1830	0.1979
LSTM	0.0062	0.0014	0.0104	0.0027
Transformer	0.0048	0.0012	**0.0074**	**0.0016**
ARN	0.0027	0.0003	**0.0056**	**0.0016**

(b)

* *Cross validation results*

got to around 240 bits. From this point on, increasing the complexity did not help reduce MSE anymore as it started to get overfitted. The second constraint for lower syntactic complexity was applied to select the solution of 249.22 bits complexity. It is still smaller than LSTM, while its MSE is 0.0031, two times better than LSTM.

3) THE SELECTED NEURONS

Equation (6) is the mathematical interpretation from the SML representation of the selected neuron for *Fovr*. Its circuit diagram is shown in Fig. 9. The overline indicates that the variable is a linear combination of the other neurons, excluding self. The neuron does not utilize internal states s. At first glance, it resembles a simple RNN neuron where the output is updated based on the previous output and the contextualized current input (1). However, ARN uses the linear combination of the outputs, excluding the output from the current nodes

$$\begin{aligned} x_1^{(t)} &= W_{(x,1)}x^{(t)} \\ v_0^{(t)} &= srelu[srelu(a_0 \odot x_1^{(t)})] + 0.5 \\ \bar{y}^{(t-1)} &= U_{(y,0)}y^{(t-1)} \\ y^{(t)} &= v_0^{(t)} \odot \bar{y}^{(t-1)}. \end{aligned} \tag{6}$$

The selected neuron for the *Sovr* signal is more complex (7). It involves two internal states s_0, s_1 and a linear combination of other peeps $\bar{s}_1^{(t-1)}$ (Fig. 10). ADATE also invented several constants for contextualizing token input

$$\begin{aligned} c_0 &= -0.11708123152147887 \\ c_1 &= 0.4690375346625194 \cdot 10^{-5} \\ c_2 &= 0.3961353427253718 \cdot 10^{-3} \end{aligned}$$

$$\begin{aligned} \bar{s}_1^{(t-1)} &= U_{(s,1)}s_1^{(t-1)} \\ v_0 &= srelu[(a_0 \odot srelu(\bar{s}_1^{(t-1)}) + \bar{s}_1^{(t-1)}) \cdot 0.5] \\ v_1 &= relu[srelu(2a_{1,2} + srelu(s_1^{(t-1)}))] \\ v_2 &= srelu(c_0 \cdot s_0^{(t-1)} - c_0) \\ v_3 &= srelu(srelu(v_2 \odot v_0)) + v_0 + y^{(t-1)} \\ a_{1,2} &= c_1 a_1 + a_2 + srelu(c_2) \\ s_0^{(t)} &= W_{(x,4)}x^{(t)} \\ s_1^{(t)} &= 0.4810137295502712 \\ y^{(t)} &= v_3 \odot v_1. \end{aligned} \tag{7}$$

C. TRAINING NETWORKS

ARN and LSTM networks were implemented in C++. The training was run using the AADC library from Matlogica [47] for automatic differentiation. Since this dataset is best analyzed using small nets, training runs faster on a CPU than on a GPU, given that a highly efficient auto diff library such as AADC is employed.

We used the Tensorflow implementation of the simple RNN model. To conduct the hyperparameters search for the simple RNN, we used the Keras Tuner [48], keeping the same search limits as for the LSTM experiment.

For the IndRNN model, we used the PyTorch implementation available on its first author's GitHub repository.[3] The hyperparameter search for IndRNN was conducted using Ray Tune [49], with the same search limits as for the other networks. In addition, since the power of IndRNN lies in the depth of the network, we also searched for the optimal number of layers from a predefined list of choices 2^i for $i = 1, 2, 3$.

We used the Hyperband algorithm found in both Keras Tuner and Ray Tune for our experiments. Hyperband is a random search method that systematically explores different configurations within a predefined schedule of iterations per configuration. Promising candidates from the initial random runs are selected for further evaluation with longer training runs [50].

The chosen version of the transformer model was provided by the authors of the transformer library.[4] We did not conduct a hyperparameter search but followed their recommended settings.

We included a traditional machine learning technique as a benchmark for comparison with deep learning methods. We selected support vector machine (SVM), a well-established algorithm in classical machine learning. It is still widely used thanks to its simplicity, reliability, and flexibility. We used the implementation of SVM from the Sklearn library [51] with radial basis function kernel.

[3][Online]. Available: https://github.com/StefOe/indrnn-pytorch
[4][Online]. Available: https://github.com/gzerveas/mvts_transformer

TABLE 4. Long Short Term Memory Written in SML

```
case
    tanh(
        lc0 InputsLC + lc0( cons( SelfOutput, OtherOutputsLC ) )
    )
of Z =>
case
    sigmoid(
        lc1 InputsLC +
        lc1( cons( SelfPeep0, cons( SelfOutput, OtherOutputsLC ) ) )
    )
of I =>
case
    sigmoid(
        1.0 + lc2 InputsLC +
        lc2( cons( SelfPeep0, cons( SelfOutput, OtherOutputsLC ) ) )
    )
of F =>
case F * SelfPeep0 + I * Z of NextState =>
case
    sigmoid(
    lc3 InputsLC +
    lc3( cons( NextState, cons( SelfOutput, OtherOutputsLC ) ) )
)
of O =>
    ( NextState, 0.0, 0.0, 0.0, O * tanh NextState )
```

TABLE 5. SML Functional Program for the Chosen Evolved Neuron for *Fovr*

```
fun f
      (
        SelfPeep0,
        SelfPeep1,
        SelfPeep2,
        SelfPeep3,
        SelfOutput,
        OtherPeepsLC,
        OtherOutputsLC,
        InputsLC
        ) =
  (
    SelfPeep3,
    SelfPeep0,
    SelfPeep2,
    SelfPeep2,
    (
      ( srelu( srelu( ( ( auxWeight( 0 ) * 0.1E1 ) *  lc1 InputsLC ) ) ) + 0.5 ) * lc0 OtherOutputsLC
    )
  )
```

We conducted eight repetitions for each algorithm to have statistically accurate results. Each repetition followed the same experimental protocol used with LSTM and ARN.

V. RESULTS AND DISCUSSION

A. PERFORMANCE ON THE DATASET

Table 3 summarizes the performance of different networks with median and standard deviations from eight repetitions for *Fovr* (a) and *Sovr* (b).

Even though IndRNNs have been shown to outperform LSTMs by a large margin in other datasets [10], they did not perform well on our dataset. Similarly, the simple RNNs also yielded poor results. Transformers surpassed LSTMs for *Sovr* but got lower test performance for *Fovr*. However, in both cases, ARNs consistently outperformed the runner-ups in terms of average MSE values. Specifically, for *Fovr*, ARNs achieved a 270% lower error rate than LSTMs, while for *Sovr*, ARNs outperformed transformers by 20%. Among all the networks, the variances for ARNs were average for both *Fovr* and *Sovr*.

B. ARCHITECTURES OF THE SELECTED NEURONS

The neuron for *Sovr*, despite its apparent complexity, is actually quite straightforward. This is because the second memory state s_1 remains constant (0.481013729550271) throughout all timesteps. After training, the auxiliary weights a_i become fixed and no longer undergo any changes at run time. This effectively transforms both v_0 and v_1 (7) into wrappers for the auxiliary weights. The circuit diagram of the neuron for *Sovr* can be simplified as shown in Fig. 11, which is equivalent to the original diagram. We can observe similarities between the

TABLE 6. SML Functional Program for the Chosen Evolved Neuron for *Sovr*

```
fun f
      (
        SelfPeep0,
        SelfPeep1,
        SelfPeep2,
        SelfPeep3,
        SelfOutput,
        OtherPeepsLC,
        OtherOutputsLC,
        InputsLC
        ) =
  (
    lc4 InputsLC ,
    0.4810137295502712,
    SelfOutput,
    SelfPeep3,
    case ( ( auxWeight( 0 ) * 0.1E1 ), SelfPeep1 ) of
      ( V1EC4DF4A, V1EC4DF4B ) =>
    case
      ( V1EC4DF4A, ( srelu( V1EC4DF4B ) + srelu( V1EC4DF4A ) ) )     of
      ( V218BDD8D, V218BDD8E ) =>
        (
(           case
            (
              (
                (
                  ( auxWeight( 1 ) * srelu( lc1 OtherPeepsLC ) ) +
                  lc1 OtherPeepsLC
                  ) *
                0.5
                ),
              -0.11708123152147887
              )           of
            ( V218BDE1C, V218BDE1D ) =>
              (
(                 case
                  case ( SelfPeep0, V218BDE1D ) of
                    ( V218BDE1E, V218BDE1F ) =>
                      (
                        srelu( V218BDE1C ),
                        ( V218BDE1F * V218BDE1E )
                        )                 of
                  ( V218BDE20, V218BDE21 ) =>
                    (
                      srelu(
                        srelu(
                          (
                            srelu( V218BDE21 ) *
                            srelu( V218BDE1C )
                            )
                          ) +
                        V218BDE20
                        ) ) +
                SelfOutput
                ) ) *
(           case ( V218BDD8D, tanh( V218BDD8E ) ) of
            ( V218BDE24, V218BDE25 ) => ( V218BDE24 + V218BDE25 ) )
          )
    )
```

neurons for *Fovr* and *Sovr*. They both employ three main operators in the same order: $\odot$, $+$, $\odot$, accompanied by auxiliary weights: a_0, 0.5 in the case of *Fovr*, and v_0 for *Sovr*. However, there are also three key distinctions that are unique to each neuron.

1) In the case of *Fovr*'s neuron, the auxiliary weight a_0 serves as an *input* gate as it is multiplied with the contextualized input $x_1^{(t)}$. It is constant for every timestep. On the other hand, *Sovr*'s neuron utilizes the auxiliary weight $v_0^{(t)}$ as a *forget gate*, as it is multiplied with the memory from the previous timestep.
2) Both neurons incorporate an *output gate*. For *Fovr*, the gate changes its value based on the information from

TABLE 7. SML Functional Program for the Simplest Program in the Pareto Front for *Fovr*

```
fun f
      (
        SelfPeep0,
        SelfPeep1,
        SelfPeep2,
        SelfPeep3,
        SelfOutput,
        OtherPeepsLC,
        OtherOutputsLC,
        InputsLC
        ) =
  (
    SelfOutput,
    SelfPeep2,
    SelfPeep0,
    SelfPeep3,
    lc3( InputsLC )
    )
```

TABLE 8. SML Functional Program for the Simplest Program in the First Cluster in the Pareto Front for *Sovr*

```
fun f
      (
        SelfPeep0,
        SelfPeep1,
        SelfPeep2,
        SelfPeep3,
        SelfOutput,
        OtherPeepsLC,
        OtherOutputsLC,
        InputsLC
        ) =
  (
    SelfPeep1,
    SelfOutput,
    ( SelfPeep0 - relu( lc2( InputsLC ) ) ),
    SelfPeep0,
    SelfPeep2
    )
```

TABLE 9. SML Functional Program for the Most Complex Program in the First Cluster in the Pareto Front for *Sovr*

```
fun f
      (
        SelfPeep0,
        SelfPeep1,
        SelfPeep2,
        SelfPeep3,
        SelfOutput,
        OtherPeepsLC,
        OtherOutputsLC,
        InputsLC
        ) =
  (
    SelfOutput,
    SelfPeep1,
    SelfPeep2,
    (
      (
        lc0( bias ) *
        srelu( lc1( cons( -0.50596074554605766E1, InputsLC ) ) )
        ) +
      SelfPeep0
      ),
    SelfPeep3
    )
```

TABLE 10. Functional Program for the Simplest Program in the Second Cluster in the Pareto Front for *Sovr*

```
fun f
      (
        SelfPeep0,
        SelfPeep1,
        SelfPeep2,
        SelfPeep3,
        SelfOutput,
        OtherPeepsLC,
        OtherOutputsLC,
        InputsLC
        ) =
  case
    case
      (
        srelu(
          (
            lc1( cons( srelu( lc1( OtherPeepsLC ) ), OtherPeepsLC ) ) *
            0.5
            )
          ),
        -0.11708123152147887
        )     of
      ( V8E4B1FD, V8E4B1FE ) =>
        ( V8E4B1FD, ( V8E4B1FE * SelfPeep0 ) )   of
    ( V1EC4DF1C, V1EC4DF1D ) =>
      (
        lc4( InputsLC ),
        0.4810137295502712,
        SelfOutput,
        SelfPeep3,
        (
          (
            srelu( ( ( V1EC4DF1C * V1EC4DF1D ) + V1EC4DF1C ) ) +
            SelfOutput
            ) *
          ( lc1( bias ) + SelfPeep1 )
          )
        )
```

the other output $U_{(y,0)}y^{(t-1)}$. In contrast, *Sovr* employs a constant gate v_1 as an output gate.

3) *Sovr*'s neuron implements a skip connection through time by utilizing the internal state $s_0^{(t)}$ as context memory.

The constant gates found in both neurons do not alter their values based on context information, unlike in LSTM and its variants. Instead, they act as regulators of the information flow, ensuring a consistent decay rate of the memory. The presence of these constant gates in both neurons supports the hypothesis regarding the **first characteristic** of our dataset. Operators can only rely on the most recent data with a consistent frequency, leading to a fixed-size long-term dependence in the time series.

C. EVOLVED ARCHITECTURES FROM THE PARETO FRONT

After surpassing the performance of LSTM, the Pareto Front of *Sovr* formed two distinct clusters (see Fig. 8). Neurons in the first cluster employed a similar strategy, utilizing internal states to delay the output by several timesteps. The primary focus of evolution within this cluster was determining the optimal number of skip connections through time. In addition, the output in these neurons was often calculated not from the current recurrent states but through skip connections, allowing for the preservation of long-term dependencies. There was a big leap in syntax complexity between the two clusters, with values jumping from 80.25 to 195.56. Following the dramatic shift in strategy, neurons in the second cluster closely resembled one another, and they could all be simplified in a similar manner as the chosen solution for *Sovr*. The evolutionary process converged on an optimal strategy that fits the characteristics of our dataset, further refining the contextualization of information and the values of the control gates. Fig. 12 illustrates the circuit diagrams of the first neuron (a), the last neuron (b) in the first cluster, the first neuron (c), and the last neuron (d) in the second cluster. It is important to note that the syntactic complexity increased as we moved along the Pareto Front. Consequently, the first neurons in each cluster represent the smallest neurons within their respective clusters. Fig. 13 demonstrates the differences in skip connections through time between the first and second clusters. As the neurons in the later cluster consistently employed the same strategy (b), it is reasonable to assume that skipping one time-step is optimal for *Sovr* in this dataset.

Despite conducting separate evolutionary processes for *Fovr* and *Sovr*, both converged toward remarkably similar optimal strategies. This finding strongly supports our hypothesis regarding the **second characteristic** of our dataset. Operators used the same input source to make distinct decisions for *Fovr* and *Sovr*. The resulting neurons further indicate similar underlying characteristics for both *Fovr* and *Sovr*.

TABLE 11. SML Functional Program for the Most Complex Program in the Second Cluster in the Pareto Front for *Sovr*

```
fun f
      (
        SelfPeep0,
        SelfPeep1,
        SelfPeep2,
        SelfPeep3,
        SelfOutput,
        OtherPeepsLC,
        OtherOutputsLC,
        InputsLC
        ) =
  case cons( SelfPeep1, OtherPeepsLC ) of
    V21ED033B =>
      (
        lc4( InputsLC ),
        0.4810137295502712,
        SelfOutput,
        lc4( V21ED033B ),
        case
          (
            (
              lc1(
                cons( srelu( srelu( lc1( V21ED033B ) ) ), V21ED033B )
                ) *
              0.5
              ),
            -0.11708123152147887
            )          of
          ( V8E4B1FD, V8E4B1FE ) =>
            (
              (
(                 case
                  case
                    let
                      fun g242C1D1A V242C1D1B =
                        ( SelfPeep0 - V242C1D1B )
                    in
                      case g242C1D1A( V8E4B1FE ) of
                        V242C219E =>
                          ( V242C219E, g242C1D1A( V242C219E ) )
                    end                     of
                    ( V1EC4DF1A, V1EC4DF1B ) =>
                      (
                        srelu( V8E4B1FD ),
                        ( V1EC4DF1B * V1EC4DF1A )
                        )                   of
                    ( V1EC4DF1C, V1EC4DF1D ) =>
                      (
                        srelu(
                          srelu(
                            ( srelu( V1EC4DF1D ) * srelu( V8E4B1FD ) )
                            )
                          ) +
                        V1EC4DF1C
                        ) ) +
                  SelfOutput
                  ) *
(                 case ( lc1( bias ), SelfPeep1 ) of
                  ( V1EC4DF4A, V1EC4DF4B ) =>
                case
                  (
                    V1EC4DF4A,
                    (
                      relu(
                        ( relu( V1EC4DF4A ) * srelu( V1EC4DF4B ) )
                        ) +
                      srelu( V1EC4DF4B )
                      )
                    )
               of
                  ( V21046F13, V21046F14 ) =>
                    srelu(
                      srelu(
                        srelu(
                          let
                            fun g2467A64B V2467A64C =
                              ( V21046F13 + V2467A64C )
                          in
                            srelu( g2467A64B( g2467A64B( V21046F14 ) ) )
                          end
                          )
                        )
                      ) )
                )
        )
```

VI. CONCLUSION

In this article, we presented recurrent neurons generated via ADATE (ARNs) for the purpose of imitation learning in controlling an industrial CNC machine based on operators' experience. The AI algorithms utilize the same data that operators have during the machining process and output two decisions for engagement rates: feed rate (*Fovr*) and velocity speed (*Sovr*). Our experiment compares the performance of ARNs to four other notable neural networks for time series analysis: simple RNNs, IndRNNs, LSTMs, and transformers. The results demonstrate the following:

1) ARNs achieve enhanced performance over other competing networks for both output signals *Fovr* and *Sovr*, exhibiting notably lower error rates. Specifically, ARNs achieve error rates that are 270% lower than LSTMs for *Fovr* and 20% lower than transformers for *Sovr*. In addition, ARNs also have lower syntactic complexity compared to LSTMs.
2) Although we conducted two evolutionary processes for *Fovr* and *Sovr* separately, both paths eventually led to comparable solutions. The outcome aligns with our second hypothesis, which proposes that operators make two distinct decisions influenced by the same dataset characteristics. This convergence suggests that the evolutionary approach effectively identified optimal structures for processing information specific to this dataset.
3) The evolutionary algorithm ADATE is capable of generating novel architectures of artificial neurons, including gate mechanisms and skip connections through time.

Our method tackles the challenge of designing neuron components within the AutoML framework. Automating this process mitigates human's limited understanding of the nature of the dataset and enhances accuracy, as demonstrated in our study. The ADATE evolutionary algorithm could generate a plethora of recurrent neuron architectures fitting for our datasets. Our approach can complement other AutoML techniques in model generation, as shown in our experiment where we incorporated Hyperband search for hyperparameter optimization.

The AI models replicating operators' experiences for safeguarding and quality control can significantly enhance the level of automation on the shop floor. They can be used for training new operators and reducing the workload on experienced operators, allowing them to allocate their labor to other tasks.

While the evolutionary algorithm required a week to generate neurons, this duration is comparable to the time and resources that machine learning experts typically invest in manually exploring various model configurations. In our research, we used data from a single month, covering only one product from MSH's production. Despite the limited data scope, the generated neurons have shown promising alignment with our initial hypotheses about operator cognition. This suggests a degree of validity in these neurons for generalizing to new data, indicating that simply retraining the neural network might suffice for new incoming data. Should there be a need to rerun the evolutionary algorithm, we anticipate a quicker process. Automating the model exploration with the evolutionary algorithm is also likely to be more cost-effective than the manual exploration of different model options. We plan to further validate the generalizability of ARN neurons with a broader dataset.

APPENDIX A
RECURRENT NEURONS WRITTEN IN SML

The appendix lists recurrent neurons written in Standard ML. Table 4 shows how LSTM is written in SML. Table 5, 6, 7, 8, 9, 10, 11 are the direct outputs from ADATE algorithm with minor modifications for formatting.

ACKNOWLEDGMENT

The authors would like to thank Mekanisk Service Halden by making available to us real data and to share their domain knowledge upon our repeated requests.

REFERENCES

[1] C. Schröer, F. Kruse, and J. M. Gómez, "A systematic literature review on applying CRISP-DM process model," *Procedia Comput. Sci.*, vol. 181, pp. 526–534, 2021, doi: 10.1016/j.procs.2021.01.199. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050921002416

[2] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[3] E. Real, C. Liang, D. R. So, and Q. V. Le, "AutoML-zero: Evolving machine learning algorithms from scratch," in *Proc. 37th Int. Conf. Mach. Learn.*, 2020, Art. no. 742.

[4] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013, doi: 10.1109/TPAMI.2013.50.

[5] B. Lim and S. Zohren, "Time-series forecasting with deep learning: A survey," *Philos. Trans. Roy. Soc. A, Math. Phys. Eng. Sci.*, vol. 379, no. 2194, Feb. 2021, Art. no. 20200209.

[6] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012.

[7] X. He, K. Zhao, and X. Chu, "AutoML: A survey of the state-of-the-art," *Knowl.-Based Syst.*, vol. 212, 2021, Art. no. 106622, doi: 10.1016/j.knosys.2020.106622. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705120307516

[8] R. Olsson, "Inductive functional programming using incremental program transformation," *Artif. Intell.*, vol. 74, no. 1, pp. 55–81, 1995.

[9] L. V. Magnusson, J. R. Olsson, and C. T. T. Tran, "Recurrent neural networks for oil well event prediction," *IEEE Intell. Syst.*, vol. 38, no. 2, pp. 73–80, Mar./Apr. 2023, doi 10.1109/MIS.2023.3252446.

[10] S. Li, W. Li, C. Cook, C. Zhu, and Y. Gao, "Independently recurrent neural network (IndRNN): Building a longer and deeper RNN," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognition*, pp. 5457–5466, 2018.

[11] G. Zerveas, S. Jayaraman, D. Patel, A. Bhamidipaty, and C. Eickhoff, "A transformer-based framework for multivariate time series representation learning," in *Proc. 27th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, New York, NY, USA, 2021, pp. 2114–2124, doi: 10.1145/3447548.3467401. [Online]. Available: https://doi.org/10.1145/3447548.3467401

[12] A. Graves, *Supervised Sequence Labelling With Recurrent Neural Networks*. Berlin, Germany: Springer, 2012.

[13] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," *Diploma thesis, Technische Universitdt Munchen*, 1991.

[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, pp. 1735–1780, Dec. 1997, doi 10.1162/neco.1997.9.8.1735.

[15] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 10, pp. 2222–2232, Oct. 2017, doi: 10.1109/TNNLS.2016.2582924.

[16] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, pp. 2342–2350.

[17] K. Cho et al., "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014. [Online]. Available: https://arxiv.org/abs/1406.1078

[18] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014. [Online]. Available: https://arxiv.org/abs/1412.3555

[19] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2016, *arXiv:1409.0473*.

[20] A. Vaswani et al., "Attention is all you need," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 6000–6010.

[21] A. Zeng, M. Chen, L. Zhang, and Q. Xu, "Are transformers effective for time series forecasting?" in *Proc. AAAI Conf. Artif. Intell.*, 2023.

[22] H. Zhou et al., "Informer: Beyond efficient transformer for long sequence time-series forecasting," in *Proc. AAAI Conf. Artif. Intell.*, 2021, pp. 11106–11115.

[23] N. Wu, B. Green, X. Ben, and S. O'Banion, "Deep transformer models for time series forecasting: The influenza prevalence case," 2020, *arXiv:2001.08317*.

[24] T. Mikolov, A. Joulin, S. Chopra, M. Mathieu, and M. Ranzato, "Learning longer memory in recurrent neural networks," 2014. [Online]. Available: https://arxiv.org/abs/1412.7753

[25] V. Campos, B. Jou, X. Giro-i Nieto, J. Torres, and S.-F. Chang, "Skip RNN: Learning to skip state updates in recurrent neural networks," 2017. [Online]. Available: https://arxiv.org/abs/1708.06834

[26] S. Chang et al., "Dilated recurrent neural networks," 2017. [Online]. Available: https://arxiv.org/abs/1710.02224

[27] E. Grave, A. Joulin, and N. Usunier, "Improving neural language models with a continuous cache," 2016. [Online]. Available: https://arxiv.org/abs/1612.04426

[28] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," 2014. [Online]. Available: https://arxiv.org/abs/1410.5401

[29] R. Olsson and A. Løkketangen, "Generating meta-heuristic optimization code using adate," *J. Heuristics*, vol. 16, pp. 911–930, 2010.

[30] R. Olsson, C. Tran, and L. Magnusson, "Automatic synthesis of neurons for recurrent neural nets," 2022, *arXiv:2207.03577*.

[31] S. Newman, A. Nassehi, R. ImaniAsrai, and V. Dhokia, "Energy efficient process planning for CNC machining," *CIRP J. Manuf. Sci. Technol.*, vol. 5, no. 2, pp. 127–136, 2012, doi: 10.1016/j.cirpj.2012.03.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1755581712000223

[32] S. Tajima and B. Sencer, "Global toolpath smoothing for CNC machine tools with uninterrupted acceleration," *Int. J. Mach. Tools Manufacture*, vol. 121, pp. 81–95, 2017.

[33] B. Li, H. Zhang, P. Ye, and J. Wang, "Trajectory smoothing method using reinforcement learning for computer numerical control machine tools," *Robot. Comput. Int. Manuf.*, vol. 61, 2020, Art. no. 101847, doi: 10.1016/j.rcim.2019.101847. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0736584519300419

[34] L. Wang, X. Yuan, H. Si, and F. Duan, "Feedrate scheduling method for constant peak cutting force in five-axis flank milling process," *Chin. J. Aeronaut.*, vol. 33, no. 7, pp. 2055–2069, 2020, doi: 10.1016/j.cja.2019.09.014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1000936119303449

[35] T. Mohanraj, J. Yerchuru, H. Krishnan, R. Nithin Aravind, and R. Yameni, "Development of tool condition monitoring system in end milling process using wavelet features and Hoelder's exponent with machine learning algorithms," *Measurement*, vol. 173, 2021, Art. no. 108671, doi: 10.1016/j.measurement.2020.108671. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0263224120311830

[36] D. F. Hesser and B. Markert, "Tool wear monitoring of a retrofitted CNC milling machine using artificial neural networks," *Manuf. Lett.*, vol. 19, pp. 1–4, 2019, doi: 10.1016/j.mfglet.2018.11.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2213846318301524

[37] L. Moreira, W. Li, X. Lu, and M. Fitzpatrick, "Supervision controller for real-time surface quality assurance in CNC machining using artificial intelligence," *Comput. Ind. Eng.*, vol. 127, pp. 158–168, 2019, doi: 10.1016/j.cie.2018.12.016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835218306211

[38] M. I. Ahmad, Y. Yusof, M. E. Daud, K. Latiff, A. Z. A. Kadir, and Y. Saif, "Machine monitoring system: A decade in review," *Int. J. Adv. Manuf. Technol.*, vol. 108, pp. 3645–3659, 2020, doi: 10.1007/s00170-020-05620-3.

[39] W. Luo, T. Hu, C. Zhang, and Y. Wei, "Digital twin for CNC machine tool: Modeling and using strategy," *J. Ambient Intell. Humanized Comput.*, vol. 10, pp. 1–4, 2019.

[40] W. Sakarinto, H. Narazaki, and K. Shirase, "A decision support system for capturing CNC operator knowledge," *Int. J. Autom. Technol.*, vol. 5, no. 5, pp. 655–662, 2011, doi 10.20965/ijat.2011.p0655.

[41] CustomPartNet, "Speed-feed-turning," 2008. Accessed: Apr. 13, 2023. [Online]. Available: https://www.custompartnet.com/calculator/turning-speed-and-feed

[42] M. Törngren and U. Sellgren, "Complexity challenges in development of cyber-physical systems: Essays dedicated to Edward A. Lee on the occasion of his 60th birthday," *Lecture Notes Comput. Sci. (Including Subseries Lecture Notes Artif. Intell. Lecture Notes Bioinf.)*, pp. 478–503, 2018, doi: 10.1007/978-3-319-95246-8_27.

[43] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, *Efficient BackProp*. Berlin, Germany: Springer, 2012, pp. 9–48.

[44] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, "On empirical comparisons of optimizers for deep learning," 2020, *arXiv:1910.05446*.

[45] M. Wooldridge, *An Introduction to MultiAgent Systems*. Chichester, U.K.: Wiley, 2009.

[46] D. A. van Veldhuizen and G. B. Lamont, "Evolutionary computation and convergence to a pareto front," in *Proc. Late Breaking Papers Genetic Program. Conf.*, J. R. Koza, Ed., 1998, pp. 221–228,. [Online]. Available: http://www.lania.mx/~ccoello/EMOO/vanvel2.ps.gz

[47] MatLogica LTD, "HPC AAD-compiler C++software library user guide," Oct. 31, 2022. [Online]. Available: https://matlogica.com/

[48] T. O'Malley et al., "Kerastuner," 2019. [Online]. Available: https://github.com/keras-team/keras-tuner

[49] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," 2018, *arXiv:1807.05118*.

[50] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6765–6816, Jan. 2017.

[51] F. Pedregosa et al., "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.