Hieroglyphically Supervised Bioinspired Visioneural Controller

Dan Berco,* Chih-Hao Chiu, and Diing Shenp Ang*

Unlike an intelligent microcontroller, biologic neural networks maintain active control throughout their operation, while the former usually needs to be disengaged during maintenance or software updates. Ideally, artificial intelligent control systems incorporate an incremental learning capability, without having the need to be retrained offline. Moreover, some hardware-based neural networks (HNNs) rely on complementary software simulators during training. Such approaches decrease their potential to implement real-time adaption, in a similar manner to their biologic counterparts. Herein, an independent, bioinspired robotic vision system with an incremental learning ability is presented. The system is trained and operated by associating instructions with hieroglyphic symbols and used to maneuver a robotic vehicle. A neural processor, capable of incremental online learning, forms the basis for the decision-making flow. The entire range of functionality, starting from training, data storage, and up to controlling, is demonstrated herein.

1. Introduction

When incorporated into intelligent control systems, hardware-based artificial neural networks (HNNs) present a significant advantage over their software counterparts, especially in terms of speed. The most ubiquitous approach for the implementation of convolutional neural networks (CNNs), consisting of multilayer perceptrons, is with software algorithms executed over von Neumann machines. These are used in a wide variety of applications such as image classifications. Such algorithms are serial in nature, meaning that a central processing unit (CPU) is used to execute them in a sequential manner. As CNN operations involve a large number of vector-matrix multiplications, application-specific integrated circuits (ASICs) were later used to implement parallel HNNs. This significant merit did not come without a cost. The main drawback was, and still is, the complexity of the design flow. In order for the frontend, high-level design to succeed, strict criteria must be met to ensure timing convergence of the backend implementation. This in turn led to field-programmable gate arrays (FPGAs) being configured as HNNs. FPGA platforms offer more flexibility when compared with ASICs, at the cost of lower performance and inferior gate count. Due to these limitations, some FPGA-based HNNs had to rely on low-precision arithmetic implementations while ASICs could end up having a very large gate count. As a workaround, several HNNs used offline training schemes using dedicated software. Obviously such dependencies limited their ability to operate in an autonomous fashion or undergo incremental learning while being kept online.

In recent years, two approaches were developed for the implementation of bioinspired robotic vision. The first was based on event-triggered sensor arrays and the second used specialized software. Using the first approach, spiking cameras were demonstrated as being able to capture fast temporal changes. These apparatuses rely on specialized ASIC implementation with all the associated complexities. As for software algorithms, they were usually developed in accordance with the two visual pathway theory which states that the visual cortex can be split into dorsal and ventral streams. These algorithms were built around a CNN, consisting of a hierarchical feed-forward data processing path, containing two bracketed pairs of convolution operator, followed by a pooling layer. However, they were
unable to successfully mimic some visual perception aspects common in biologic organisms such as detail preservation. More recently, bioinspired, artificial smart retinas based on a 2D array paradigm of photoelectric logic gates were also suggested.\textsuperscript{[30–33]} Followed by a three-input material, nonimplication and logic conjunction gate constructed out of two memristors and a pull-down resistor was demonstrated that may be used to perform in situ image compression.\textsuperscript{[34]}

The biologic retina consists of light-sensitive rods and cones. Photons (with wavelengths in the visual range) stimulate them to produce excitations that are converted into electric signals and propagated by synaptic junctions in bipolar cells.\textsuperscript{[35–37]} Simplistically speaking, bipolar cells may be classified into two main categories: off center and on center. An off-center cell displays a depolarized response under illumination that changes into a depolarized one in darkness. An on-center cell shows a depolarized response under illumination and hyperpolarized in darkness. In this manner, center-surround configurations, where one type of cell is surrounded by opposite ones, allow for better accuracy and resolution in visual perception. The approach presented in Bero et al.\textsuperscript{[38–40]} attempted to mimic this functionality by divergence acquisition of a pixel map, as opposed to conventional paradigms that usually rely on pixel-array snapshots as inputs to CNNs.

This work features an intelligent, bioinspired, robotic visioccontrol system with incremental learning ability. It is fully autonomous in the sense that it does not rely on external means such as software simulators for learning. The system is composed of three main modules. The first is a camera-vision module (VM) that effectively implements bioinspired robotic vision, where visual data are converted into bit information and communicated to a neural processor (NP, second module). The NP forms the basis for decision-making HNN. It is implemented over a commercial FPGA platform using less than 45 K logic elements and is capable of incremental, online learning. Visual images are processed by the VM–NP interaction in a similar manner to the optoelectronic signal transduction that occurs within the mammalian retina. The system is trained by associating an instruction set with hieroglyphic symbols presented to the VM. Once trained, these symbols are used to issue instructions to a control module (CM) and maneuver a robotic vehicle. The CM is based on a commercial ESP32 microcontroller (MCU) and used to activate two small electrical engines, spinning the wheels of a robotic vehicle. Dedicated blocks in the NP enable serial communication of both data streams and instructions to either the VM or CM. This manuscript presents the control system’s whole range of operation starting from training, data storage, user interfacing, and up to vehicle control.

2. Results and Discussion

2.1. Architectural Concept

Figure 1 depicts the architecture of the visioccontrol system along with the experimental setup used to both train and evaluate its performance. Hieroglyphic pattern generation method is shown in Figure 1a. For this purpose, a Bluetooth terminal application running on an Android device was used. This app interacted with an MCU that in turn operated an 8 × 8 light-emitting-diode (LED) array.\textsuperscript{[38]} Each displayed hieroglyphic image was assigned a predefined instructional interpretation. Once trained, the neural controller was intended to execute those instructions in response to redisplaying the corresponding hieroglyphs.

The VM unit incorporates a minicamera. In addition, its main components are shown in Figure 1b. Bioinspired image acquisition, as executed by this module, will be detailed in a dedicated section. The postprocess image is translated to an input vector that is transmitted to the NP over a universal asynchronous receiver–transmitter (UART) protocol by a UART-DMA (UART to direct memory access, UDMA) block through the general-purpose input–output (GPIOs). As for the NP, it was implemented over a commercial DE-10 board, as shown in Figure 1c. The board incorporated an FPGA chip along with circuitry designed to program it. It also contains GPIO pins, switches, toggle keys, and seven-segment display modules. An NP lies at the heart of this system. This NP was configured as a HNN with four output nodes, representing a binary code from 0 to 15 that was assigned to tags. Vector-matrix operations were implemented by an arithmetic representation of 20-bit fixed-point integers. The main constraint was the board’s 50 K logic element count limitation. There is a tradeoff between computation accuracy and system size (gate count) that will be discussed later on. Lower accuracies, related to lesser bit representations, may not allow the backpropagation algorithm to settle properly. For instance, in this work, it was discovered that integer widths of 16 bits or less failed to accomplish convergence.

The peripheral blocks shown within the DE-10 frame in Figure 1a allow the NP to function independently without relying on external modules. The first block (marked I/F) executed the serial interface protocol. It contains a synchronizer to ensure the proper transfer of image information, image tag, and other flags from the UART clock domain to the system’s 50 MHz clock domain. In addition, it has a specialized pooling layer that is used to reduce the input’s data size (once fully received) through normalized threshold operations over image partitions.\textsuperscript{[41]} Another block allows for unconstrained DMA to the onboard synchronous dynamic random access memory (SDRAM). Training vectors, defined by combined pooled image information and their corresponding tags, are automatically stored into the memory. The DMA is responsible for properly executing read, write, and shuffle instructions. Such instructions are invoked by a general control state machine (STM) as part of the training or testing procedure. Interaction between the STM and the DMA is accomplished through handshake signaling. The NP is switched between “test” and “train” modes automatically by this STM. Six seven-segment displays (7-Seg) are used to show the overall training progress by displaying the convergence error. During testing, the 7-Segs display the NP’s output. Data packets between the VM and NP were transferred using a serial communication protocol through a serial interface.

The structure of the CM is shown in Figure 1d. Electric motor control was done by yet another MCU that received its instruction from the NP over dedicated GPIO lines. Those instructions served as entries for an internal lookup table that stored a set of motor-step-like orders delivered to an H-bridge electric motor driver. For example, to turn right, the left wheel was driven
forward and right wheel driven backward for a preset number of consecutive steps. To go forward, both wheels were driven forward (taking into account that they are mirrored). The received instructions were displayed for reference over a small, monochromatic OLED using \( \text{I}^2\text{C} \) protocol. Figure 1e includes four images that depict the four preset hieroglyphs, as displayed over the 8 \( \times \) 8 LED array, where each device produced an irradiation power density of \( \approx 20 \text{ mW cm}^{-2} \). As mentioned previously, these hieroglyphic instructions were assigned a predetermined interpretation that later translated to motor stepping sequences by the CM. The hieroglyphs along with their corresponding interpretation are 1) step back and turn right 90° (\( \downarrow \)), 2) step back and turn left 90° (\( \downarrow \)), 3) step forward and turn right 90° (\( \uparrow \)), and 4) step forward and turn left 90° (\( \uparrow \)). Two pictures showing the entire setup and its components are given in Figure 1f,g. The first is a side-view photo and the second a top-view one.

### 2.2. Serial Communication

This section discusses the serial communication method and protocol as implemented in this work. An overview of the hardware configuration is given in Figure 2. The figure details the communication blocks of the VM in Figure 2a and the ones implemented over the FPGA in Figure 2b. The VM was based on an off-the-shelf MCU that has three inherent UART interfaces that share two UDMA controllers. This UDMA implements an encoding format of separator-wrapped data chunks. Encoders are therefore used for adding separators before and after each data
chunk, as well as for replacing data that are the same as separators by special-character sequences. The decoder performs the inverse operation, meaning that it is used for removing separators before and after data, as well as for replacing the said special-character sequences with separators. In addition, there may be multiple consecutive separators marking the beginning or end of data chunks.

In Figure 2b, at the NP’s end, an asynchronous interface module (Serial I/F) was implemented and associated with two pins on the board’s GPIO. Those pins were wired to transmit and receive lines (Tx/Rx) of the VM’s GPIO. Data streams were latched by the I/F and passed to a Pooling layer that reduced their size by a factor of \(1/n^2\), where \(n = 1, 2, \) or 3. Pooling’s output was then threshold normalized and latched by a DMA module. Once complete, an internal STM (in the DMA) triggered a memory write operation and both image data and tag were stored in the SDRAM. The memory was filled as a first-in-first-out (FIFO) queue. A register pointer was used to store the highest valid address for reference during training.

Figure 2c elaborates the streaming protocol for visual information from the VM to the NP. The same protocol was used for training and testing. Each transmission began with a start byte, indicating the beginning of a new vector (i.e., image data and tag). It was followed by an ASCII byte representing the tag (“0”, “1”, “2”, etc.) that was assigned by the VM. The following bytes were the data bytes, starting from the most significant byte (MSB) and ending with the least significant one (LSB). A stop byte was used to seal each sequence. Once four different hieroglyphs were transmitted, a finish byte was sent by the VM and training automatically initiated by the control STM in the NP (Figure 1c). Note that the start and stop bits inherent to the UART protocol are omitted for simplicity.

2.3. Parallel Neural Processor

Artificial neural entities may be implemented either as software programs or dedicated electronic circuitry. In the first case, they are realized as data structures over which an algorithm is executed, usually in a serial manner. As for HNNs, calculations are ideally intended to take place in parallel. Regardless of the approach, artificial neurons are linked in the form of a directed graph. In the software’s case, the learning process is incorporated into the algorithm, while in the hardware’s case, it may be directly infused into the entity. This latter method was adopted in this work and its concepts are illustrated in Figure 3. Computations are based on 20-bit, fixed-point integers, as mentioned previously.

Figure 3 details blocks that function as registers (Reg) and circles that represent combinational logic operations. As seen in the figure, there are two main data paths. The forward path produces the weighted output evaluation or NP’s output. The backpropagation path is responsible for updating the weight register bank during supervised learning. In the forward path, all inputs are multiplied by corresponding weights and summed in a completely parallel manner. This sum is fed into an activation circuit and latched by an output register. Each neuron thus
generates a weighted output as a function of its inputs and a non-
linear activation function. The implementation of the activation
function is discussed in the supplementary content.

While the forward path is active during both training and
normal operation (or testing), backpropagation was enabled only
during training. The so-called synaptic weights may therefore be
updated only then. The backward path initially evaluated the error
between the generated output and the required one, as stored in
the target register. This target was directly derived from the tag and
updated with every new image in the training dataset. The error,
sometimes referred to as cost function, was stored in the Delta
register. It was multiplied by a fixed learning rate value, in addition
to the input value. The result was then used to update a weight
change register. It is more likely that this process will converge
if weight changes are not done too abruptly. For this reason, a pre-
viously estimated weight change was multiplied by a constant
(Momentum) value close to (and lower than) 1. Therefore, a newly
generated weight change will not be too far off the previous one.
This approach allows for a better-behaved and smoother com-
putation. Finally, the existing synaptic weight was updated by adding
the calculated weight change as $w_{ij} = w_{ij} + \Delta_{ij}$. This process was
repeated for every image in the dataset stored in the SDRAM.
Once all the memory content has been used, a single training
epoch was said to be complete and the total error accumulated
and evaluated. To better assist with convergence, the memory con-
tent was randomly shuffled after each epoch, so the training vec-
tors were not present using the same order in the next epoch.

Different register instances along with their corresponding bit
counts are elaborated in Table 1.

### 2.4. NP Logic Element Count

Parallel and distributed computing is an important branch in the
evolution of computer architectures. In some implementations,
distributed systems are based on large clusters of relatively sim-
ple processors. Heavy computational tasks are divided into
threads and distributed among those units. This approach has
also found its way to the implementation of multicore CPUs.
It is therefore not unlikely that a sub-branch in neural processing
should take this path as well. Such a trend dictates the use of
small HNNs joined together to perform a parallel task that would
otherwise require a very large CNN.

This section investigates the implementation size of the NP
and peripheral circuitry in terms of logic and register element
count (or LUT). Overall size is determined by the backend flow,
after analysis and elaboration. This compilation process takes
into account both the hardware and constraints, to achieve proper
timing closure. In addition, the element count cannot exceed
50 K in DE-10. Actually, this upper limit cannot be reached,
and in practice 45 K is a more realistic number for implementa-
tions. The main reason is that the built-in router will most likely
be unable to converge for such high densities. Other than that,

### Table 1. Register element count and associated bit size for a four-neuron hidden layer with 36-bit pooling:

| Register       | Instances | Bit count | Total |
|----------------|-----------|-----------|-------|
| Learning rate  | 1         | 20        | 20    |
| Momentum       | 1         | 20        | 20    |
| Output         | 4         | 20        | 80    |
| Target         | 4         | 20        | 80    |
| Delta          | 4         | 20        | 80    |
| Input          | 1         | 36        | 36    |
| Weight $[w_{ij}]$ | $4 \times 36$ | 20 | 2880 |
| Change $[\Delta_{ij}]$ | $4 \times 36$ | 20 | 2880 |
| Total          | 303       | 6076      | 6076  |
timing closure would become exponentially difficult, as more routing resources would be allocated to meet the constraints.

Figure 4a shows the dependency of the NP’s element count on the hidden layer size. It is evident that both logic and register numbers increased rapidly as the hidden node number grew. This figure grew by \( \approx 10^K \) for an increase in +2 in node count. This behavior is expected as parallel summation and multiplication modules that were intentionally incorporated into each artificial neuron contributed the most to this growth. The reader may recall that within the hidden layer, this figure is in direct relation to the input vector size. Therefore, increasing the hidden layer count by +1 resulted in adding many multiplication modules. Of course, the number of neurons in the hidden layer determines the robustness of the HNN (the more the better) at the cost of overall size. It was shown in this work that the HNN can successfully identify nine out of ten patterns with only two hidden nodes. This rate grew as the number of hidden nodes increased.

An alternative approach, where these modules are shared (using multiplexers) between layers, or even between neurons at the same layer, may be employed at the cost of reduced parallelism. However, the computation time for each layer would increase by roughly two orders of magnitude, as the number of required clock cycles will grow as well. In the figure, values larger than 50 K were recorded after the analysis and synthesis stage. The compiler is able to provide a reliable and true gate count based on the design. Of course, the complete flow could not be accomplished as these numbers surpassed the capacity of the FPGA. Therefore, only the element count was extracted and used.

On the other hand, the number of logic elements required to implement the peripheral circuitry grew linearly, at a much slower rate, with the pooling layer’s output size, as shown in Figure 4b. Larger sizes corresponded to a lower pooling factor. For example, an output that is similar to the image bit count corresponds to a factor of 1. This trend may be explained by the fact that the pooling layer is the more dominant block, in terms of size, in the peripheral circuitry. Finally, the NP’s size dependency on the fixed-point integer representation width was studied and is given in Figure 4c. The overall element count grew super linearly in a similar manner to Figure 4a. It is apparent that the logic count increased by \( \approx 100 \) for +10 bits. This is probably because this bit count affects all the nodes in an artificial neural network at the same rate (i.e., all neurons’ implementation will inflate similarly regardless of their placement in the network).

Figure 4d illustrates the differences between two major approaches in the implementation of hardware-based artificial neurons. The first shows parallel processing that was adopted in this work. In this approach, the design is flattened and a high level of parallelism is achieved. Input vector and weight matrix multiplication may be done within a single clock cycle and summed up. The main disadvantage is the large amount of hardware required to implement it. On the other hand, serial processing requires a single multiplier and a relatively simple adder. In this case, a multiplexer is used to select one input port and its corresponding weight value at a time. They are then multiplied during a single cycle and added to the overall sum. In this manner, the number of cycles required to process each and every layer can grow by several orders of magnitude, depending on the input vector’s size.

2.5. Incremental Learning

Incremental learning was demonstrated in this work by uploading an increasing pattern count to the HNN and repeating the
training process while keeping the system online. This interaction was based on subsets of 10, 20, 30, and 40 processed images from the MNIST database[42] and carried out by a personal computer. Hieroglyphs presented to the vision system are transuded into bit streams and transmitted over the serial link. Those bits form input vectors that are then used to train the hardware-based NP. MNIST images are also translated into streams of bits which serve the exact same purpose. Therefore, from the NP’s point of view, these vectors consist a whole training set, no matter whether they originate from a VM or a computer. The details are discussed in Supporting Information.

This experiment was conducted by first loading the training set as the system was configured in training mode. The pushbutton KEY[1] was used to manually initiate a start command (bypassing the Control STM in Figure 1a). Once training was completed, the system was switched to test mode. KEY[1] was then used to manually send read instructions to the DMA. The memory’s output was fed into the NP’s input and the NP’s output compared with the image tag using the 7-Seg displays. These training results are summarized in Figure 5. The pattern count was steadily increased from 10 to 40 in steps of 10. Figure 5a-d shows the number of epochs needed to train the NP up to an error level below $2^{-5} = 0.03125$. An epoch was defined by cycling through all the training patterns and evaluating the total error at the end. As apparent from the figures, the error was successfully reduced to the predetermined level in every figure. The typical number of required epochs was between 100 and 300. Once converged, the NP was tested using sequential read operations that cycled through the memory content as mentioned previously. In all cases, the NP correctly identified all the patterns.

The initial error in Figure 5a–d is similar in all cases as a similar amount (10) of new patterns is added in each step. Existing vectors do not contribute to this initial error as they underwent training in the previous step and the network weights were already set accordingly. On average, it takes about 220 epochs to complete each training session. The number of epochs has an associated variance as the algorithmic search path for a minimum depends on the randomness of initial conditions as new vectors are introduced into the calculation. The robustness of the gradient decent process is apparent in Figure 5. Temporal spikes in the error function, usually associated with local minima, were quickly corrected as convergence kept going until a global minimum was reached.

The training process in Figure 5b shows more spikes when compared with the others. Such spiking may be explained by the randomness of initial conditions and convergence of the backpropagation flow along with its dependence on the training set. In the beginning, weights (stored in the register bank) are randomly initialized to either positive or negative values. In addition, the initial error is calculated by summation over the entire set. This error is therefore a function of specific image bits that are fed as inputs to the CNN. Training is performed by a mathematical algorithm that is looking for a satisfactory minimum in a multidimensional function (with weights serving as variables). The search path is thus determined by the backpropagation process. As initial conditions change (new vectors with different random weights), the search may either be faster (“lucky”) or slower. Furthermore, it may end up following a relatively “noisy” route, as shown in Figure 5b.

Incremental offline training is by no means limited to supervised learning through personal computer interfacing. As far as the HNN is concerned, data streams that passed via the serial interface serve the same purpose regardless of their origin. These chunks of information are used to build the incrementally
growing training set during the learning session. In this section, subsets of the MNIST image bank were chosen to construct the training reservoir as a proof of concept. In a case where hieroglyphic-based incremental training mode is used, under uncertain light-source environmental conditions, an adjustable threshold may be used to counter these effects, as will be detailed in the next section.

2.6. Bioinspired Machine Vision

The principles of the current bioinspired machine vision algorithm were presented in previous publications.[38–40] This section will discuss the main concept as implemented in this work. The reader is advised to consult those papers for further details. As mentioned previously, hieroglyphic training was achieved through the visual path of the system. Initially, images were captured in Quarter Video Graphics Array (QVGA) format with a frame resolution of 320 × 240 bits. This image bit vector was transformed to a grayscale data structure (Vo,ij). Normalization was done over 10 × 10 bit chunks, yielding a 32 × 24 matrix, which was then edge trimmed to a 24 × 18 matrix (Qkl) containing 544 bits. This normalization-abstraction process took into account the so-called brightness of nearest neighbors, in an attempt to mimic the “center-surround” interactions in biological retinas.

Initially, a pooling average was done over each chunk (indexed as k = 1.32 and l = 1.24).

\[ V_{av,kl} = \frac{1}{f_j} \sum_{c,j=1}^{90} V_{o,ij} \]  
(1)

A continuous saturation intensity figure \( V_{AVG} \) was calculated by moving average over all the chunks in consecutive images.

\[ V_{AVG} = \sum_{m=1}^{k+l} V_{av,m+1} + (m-1) V_{av,m-1} \]  
(2)

Each output was compared to a preset threshold \( V_{TH1} \) and a binary digit \( W_{kl} \) was assigned to the chunk to establish an activation threshold.

\[ W_{kl} = 1 \text{ if } V_{av,kl} \geq V_{TH1} \text{; 0 otherwise} \]  
(3)

These binary bits were added (summation over all matrix elements) and once the summation surpassed a defined sensory limit, image latching and incremental tag assignment took place.

\[ \sum_{k,l} W_{kl} \geq \text{sensory limit} \]  
(4)

This approach was used to determine a saturation-dependent threshold and mimic sensory relaxation found in living organisms’ retinas by constantly resetting these average levels. In addition, it was used to identify new patterns during training, as it contained time periods where no illumination was applied (spacing between hieroglyphs).

The unit-less matrix \( Q_{kl} \) was derived from these levels using the Arrhenius equation. A normalized activation parameter was generated by application of the differential Laplace operator with cyclic boundary conditions.

\[ Q_{kl} = \exp \left( \frac{1}{V_{AVG}} \nabla^2 V_{av,kl} \times \Delta x \Delta y \right) \]  
(5)

where \( \Delta x, \Delta y = 1 \). The 1D vector \( q_n \) (transmitted as an input to the NP) was determined based on the elements of \( Q_{kl} \) by a process of threshold \( \alpha \) binarization.

\[ q_n = 1 \text{ if } Q_{kl}^{edges} \geq \alpha ; \quad 0 \text{ otherwise} \]  
(6)

In this manner, the VM remained in a standby until exposed to a predetermined, minimal amount of irradiation. Once this limit was reached, the VM was triggered to perform the above-detailed calculation. The overall sensitivity may be modified in real time to cancel out environmental bias by adjusting \( V_{TH1}^{exp} \) and \( \alpha \). The bias is a direct result of ambient lighting that affects both brightness and saturation levels in the acquired images. Different \( V_{TH1}^{exp} \) and sensory limit may be used to indicate whether a symbol is being presented to the VM with reference to different background conditions. The threshold may therefore be adjusted in real time by constantly averaging brightness levels during idle periods.

2.7. Training and Testing

In this section, the NP was configured using four hidden neurons. Training was carried out by associating a predefined instruction set with four hieroglyphic images as discussed previously. These images were presented to the VM which sampled them at a rate of \( \approx 3–5 \) fps. Each hieroglyph had a tag associated with it. The threshold level was preset in this case to 110%. Captured image bits were therefore set to a high logic level once \( V_{avg,kl} \) surpassed \( 1.1 \cdot V_{AVG} \), as defined in Equation (3). After all the images were captured, NP training was initiated automatically until the overall error fell below \( 2^{-3} = 0.03125 \). Each training epoch was set by cycling through all the patterns and evaluating the total error at its end. Once training was completed, the NP switched into operating mode and pattern recognition was used to issue control instructions to the CM.

Figure 6 depicts a top-view image of the test setup. An Android Smartphone running a Bluetooth terminal application 1) was used to transmit control sequences encoded as number series (i.e., “1234”). Each number in the series was correlated to a hieroglyph in the instruction set. Control sequences were received by the first MCU (Figure 1a) that used them to display a corresponding hieroglyphs over the LED array (2). This array was placed vertically in front of the VM, as shown in Figure 6. In this manner, hieroglyphic sequences were displayed according to numbers in each series. For example, sending “1234” resulted in displaying “\( \gamma \) γ γ γ” while sending “4321” resulted in displaying “\( \gamma \gamma \gamma \gamma \gamma \).” The VM sampled and processed those hieroglyphs into tag/vector combinations which were transmitted over the serial link to the NP (3). A control state machine (Figure 1c) was designed to be in training mode as the NP ‘woke-up’. In this manner, the first vector set received after “reset” was considered as the training set and stored in the internal SDRAM (Figure 1c). As the VM sampled images a few times per second, each hieroglyph was
associated with several vectors (each hieroglyph was displayed for roughly 2 s).

Once the entire hieroglyphic instruction set was associated with vectors, the STM automatically initiated training. This computation took about 3 s. As training was completed, the first four 7-Segment modules (left hand side of the row) showed the 16 LSB of the 20-bit fixed point integer overall error value “00E8” (upside-down view). Testing was performed by submitting all the images in a random order via the same Bluetooth channel and observing the NP's response. This response was displayed over the 7-Segment modules as numbers that were supposed to match the ones sent over Bluetooth as a test series. The NP successfully recognized them all and issued the proper signaling to the second MCU (4) that in turn manipulated the motor driver (5). These instructions were also displayed over the OLED (7) for reference.

3. Conclusion

In summary, this work presented an autonomous bioinspired, visiocontrol system with incremental learning capability. The architecture and operation of its three main components were demonstrated and discussed. The main functional blocks are bioinspired VM, NP, and electric-engine CM. A VM converts visual data into bit information and communicates it to the NP during both training and normal operation. The NP is based on HNN and is capable of incremental, online learning. The system was trained by associating an instruction set with hieroglyphic symbols presented to the VM. Once trained, these symbols were used to issue instructions to the CM and maneuver a robotic vehicle. The CM is used to activate two small electrical engines spinning the wheels of a robotic vehicle. Dedicated blocks in the NP enable serial communication of both data streams and instructions to either the VM or CM. In this work, the system's operation, starting from training, data storage, user interfacing, and up to vehicle control, was demonstrated.

4. Experimental Section

The entire system was constructed using off-the-shelf components. The VM was implemented using an MCU, ESP32-CAM platform. Image information was captured and the bit map processed and passed to the NP module through serial communication. The NP was implemented through Verilog coding within Intel Quartus Prime development platform. Test benches were implemented using the same tool, and register-transfer-level simulations were done over Mentor Graphics Modelsim. The code was compiled within the Quartus environment and programmed through a universal serial bus (USB) blaster to a Terasic DE10-Lite board, hosting an Altera MAX-10 FPGA with a maximum capacity of 50 K logic elements and a 50 MHz internal clock. The training dataset was uploaded to board’s SDRAM through the designed serial interface and direct memory access block.

Serial communication between the VM and NP was done using UART protocol at a baud rate of 115 200 bps, through the on-board GPIO. Dedicated data packets were defined and transmitted to indicate the start and stop frames for each image pattern, as well as to mark the end of the training dataset. Each item in the training dataset was acknowledged once a new image was properly received and stored by the NP. The CM was implemented on a second MCU, ESP32 module. The control instructions (from the NP) were received through the GPIO pins and displayed on a 0.91 in., 128 × 32 monochrome OLED unit over a C communication protocol. Based on a preset pulsing sequence for each instruction, the CM operated a DRV8833 dual H-bridge motor driver (1.2 A) connected to two 3 V, 200 rpm DC motors.

Image pattern generation was done over an 8 × 8 white LED array, each with an irradiation power density of about 20 mW cm⁻². The array was operated by selecting designated bit lines and cycling the rows to a high logic level. Word and bit lines were driven by a separate ESP32 CM. Pattern instructions were supplied by the user through Bluetooth communication using an Android serial Bluetooth terminal. This serial stream was parsed by the ESP32, which was preconfigured to produce different images in response to different instructions.

Supporting Information

Supporting Information is available from the Wiley Online Library or from the author.

Acknowledgements

The authors acknowledge the partial funding support by Singapore Ministry of Education under grant MOE-T2EP50120-0003.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

D.B conceived the concept, developed the architectural methodologies, implemented the experimental setup and modeling, performed the experiments and simulations, created the visualization, and wrote the manuscript. C.H.C provided editorial comments and discussions. D.S.A obtained partial funding and provided comments.
Data Availability Statement

The data that support the findings of this study are available in the supplementary material of this article.

Keywords

artificial neural controls, artificial visual perceptions, bioinspired machine vision, bioinspired machine vision, cognitive artificial vision, robotic retinas

Received: March 9, 2022
Revised: April 28, 2022
Published online: May 29, 2022

[1] N. Morgan, in Artificial Neural Networks: Electronic Implementations, Computer Society Press Technology Series, Computer Society Press of the IEEE, Washington, D.C. 1990.
[2] G. Dundar, K. Rose, in Proc. Fifth Annual IEEE Int. ASIC Conf. and Exhibition, IEEE, NY, USA 1992, pp. 419–422.
[3] M. Yasunaga, N. Masuda, M. Yagyu, M. Asai, K. Shibata, M. Ooyama, M. Yamada, T. Sakaguchi, M. Hashimoto, in Proc. IEEE Int. Joint Conf. on Neural Networks, IEEE, NY, USA 1991, pp. 1844–1849.
[4] B. Bien, Am. Sci. 1988, 76, 561.
[5] F. Rosenblatt, in The Perceptron-A Perceiving and Recognizing Automaton, Cornell Aeronautical Laboratory, Buffalo, NY 1957.
[6] F. Rosenblatt, Psychol. Rev. 1958, 65, 386.
[7] W. Zhang, in Proc. of Annual Conf. of the Japan Society of Applied Physics, IOP, Japan 1988.
[8] W. Zhang, K. Itoh, J. Tanida, Y. Ichikawa, Appl. Opt. 1990, 29, 4790.
[9] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Nature 1986, 323, 533.
[10] M. C. Mozer, Complex Syst. 1989, 3, 349.
[11] S. Tam, B. Gupta, H. Castro, M. Holler, in Proc. IEEE Int. Conf. Syst., Man Cybern., IEEE, NY, USA 1990, pp. 701–703.
[12] H. McCartor, VLSI Artif. Intell. Neural Netw. 1991, 357.
[13] G. P. Economou, E. P. Mariatos, N. M. Economopoulos, D. Lymperopoulos, C. E. Goutis, in Proc. of the Fourth Inter. Conf. on Microelectronics for Neural Networks and Fuzzy Systems, IEEE, NY, USA 1994, pp. 287–293.
[14] S. L. Bade, B. L. Hutchings, in Proc. of IEEE Workshop on FPGA’s for Custom Computing Machines, IEEE, Piscataway, NJ 1994, pp. 189–198.
[15] D. Anguita, A. Boni, S. Ridella, IEEE Trans. Neural Netw. 2003, 14, 993.
[16] H. Ng, K. Lam, IEEE Trans. Neur. Netw. 2003, 14, 1413.
[17] Y. Maeda, M. Wakamura, IEEE Trans. Neural Netw. 2005, 16, 1664.
[18] J. Zhu, P. Sutton, Proc. Conf. Field Programm. Logic, Springer, Berlin, Heidelberg 2003, p. 1062.
[19] J. Holt, T. Baker, Proc. Int. Joint Conf. Neural Netw. 1991, 2, 121.
[20] A. W. Savich, M. Moussa, S. Areibi, IEEE Trans. Neural Netw. 2007, 18, 240.
[21] E. Säckinger, B. E. Boser, J. M. Bromley, Y. LeCun, L. D. Jackel, IEEE Trans. Neural Netw. 1992, 3, 498.
[22] U. Ramacher, W. Raab, J. U. Hachmann, J. Beichter, N. Bruls, M. Wesseling, E. Sicheneder, J. Glass, A. Wurz, R. Manner, in Proc. of 9th Inter. Parallel Process. Symp., IEEE, NY, USA 1995, pp. 774–781.
[23] M. Gschwind, V. Salapura, O. Mainsberger, in IEEE Inter. Symp. on Circuits and Systems, IEEE, Piscataway, NJ 1996, pp. 49–52.
[24] P. Lichtsteiner, C. Posch, T. Delbruck, IEEEJ. Solid-State Circuits 2008, 43, 566.
[25] C. Posch, T. Serrano-Gotarredona, B. Linares-Barranco, T. Delbruck, Proc. IEEE 2014, 102, 1470.
[26] L. Mennel, J. Symonowicz, S. Wachter, D. K. Polyushkin, A. J. Molina-Mendoza, T. Mueller, Nature 2020, 579, 62.
[27] N. K. Medathati, H. Neumann, G. S. Masson, P. Kornprobst, Comput. Vis. Image Understand. 2016, 150, 1.
[28] N. Kruger, P. Janssen, S. Kalkan, M. Lappe, A. Leonards, J. Piater, A. J. Rodriguez-Sanchez, L. Wiskott, IEEE Trans. Pattern Analysis Mach. Intell. 2012, 35, 1847.
[29] D. Cox, T. Dean, Curr. Biol. 2014, 24, R921.
[30] D. Berco, D. S. Ang, Adv. Intell. Syst. 2019, 1, 1900003.
[31] D. Berco, D. S. Ang, H. Z. Zhang, Adv. Intell. Syst. 2020, 2, 1900115.
[32] D. Berco, Y. Zhou, S. R. Gollu, P. S. Kalaga, A. Kole, M. Hassan, D. S. Ang, ACS Nano 2018, 12, 5946.
[33] D. Berco, D. S. Ang, J. Vac. Sci. Technol. B 2019, 37, 061306.
[34] D. Berco, D. S. Ang, P. S. Kalaga, Adv. Intell. Syst. 2020, 2, 2000079.
[35] S. H. DeVries, Neuron 2000, 28, 847.
[36] H. Wässle, B. B. Boycott, Physiol. Rev. 1991, 71, 447.
[37] R. H. Masland, Nat. Neurosci. 2001, 4, 877.
[38] D. Berco, D. S. Ang, Adv. Intell. Syst. 2021, 3, 2100025.
[39] D. Berco, D. S. Ang, J. Vac. Sci. Technol. B 2021, 39, 053202.
[40] D. Berco, D. S. Ang, J. Phys. D: Appl. Phys. 2020, 53, 25LT02.
[41] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, J. Schmidhuber, in Proc. of the Twenty-Second Inter. Joint Conf. on Artificial Intelligence, AAAI Press, CA, USA 2011, p. 1232.
[42] Y. LeCun, C. Cortes, C. Burges, MNIST Handwritten Digit Database, 2010, http://yann.lecun.com/exdb/mnist/ (accessed: April 2022).