A dual-memory architecture for reinforcement learning on neuromorphic platforms

Wilkie Olin-Ammentorp1,2, Yuri Sokolov and Maxim Bazhenov1,2,*

1 Department of Medicine, University of California, San Diego, CA, United States of America
2 Institute for Neural Computation, University of California, San Diego, CA, United States of America
* Author to whom any correspondence should be addressed.
E-mail: mbazhenov@health.ucsd.edu

Keywords: reinforcement learning, neuromorphic hardware, program architecture

Supplementary material for this article is available online

Abstract

Reinforcement learning (RL) is a foundation of learning in biological systems and provides a framework to address numerous challenges with real-world artificial intelligence applications. Efficient implementations of RL techniques could allow for agents deployed in edge-use cases to gain novel abilities, such as improved navigation, understanding complex situations and critical decision making. Toward this goal, we describe a flexible architecture to carry out RL on neuromorphic platforms. This architecture was implemented using an Intel neuromorphic processor and demonstrated solving a variety of tasks using spiking dynamics. Our study proposes a usable solution for real-world RL applications and demonstrates applicability of the neuromorphic platforms for RL problems.

1. Introduction

As the number of data-collecting devices increases, so too does the need for efficient data processing. Rather than require all data collected from remote devices be processed at a central location, the need for data processing to be performed in-situ is becoming a priority; this is especially true in situations where 'agents' collecting data may need to make critical decisions based on low latency (such as in self-driving cars or aerial drones). For such use cases, efficiency of data processing becomes paramount, as energy sources and physical space ('size, weight, and power') come at a premium [1].

Neuromorphic architectures provide one path toward meeting this need. Although there is no universal definition on what constitutes a neuromorphic architecture, these systems generally aim to provide efficient, massively-parallel processing schemes which often use binary 'spikes' to transmit information [2, 3]. Given that a clear definition of a neuromorphic architecture is not yet universally agreed upon, it is difficult to design a single program which can be compiled to any neuromorphic system (as is the case with standard computer architectures). But, by restricting ourselves to the massively-parallel operations which are a common feature of almost all neuromorphic systems, we can create a program which likely be adapted to any platform which meets the emerging definition of what constitutes a neuromorphic system [4]. In this work, we utilize Intel's neuromorphic processor codenamed 'Loihi' [5].

Reinforcement learning (RL) represents a native view of how biological systems learn. Instead of being trained before deployment by massive amounts of labeled data, humans and animals learn continuously from experience by updating a policy based on continuously collected data [6–9]. This requires learning to occur in situ rather than depending on slow and costly uploading of new data to a central location where new information would be embedded to the previously trained model, followed by downloading the new model to the agent.

Toward these objectives, we describe a high-level system for carrying out RL tasks which is inspired by simplified principles of biological computation. In particular, we apply a basic aspect of complementary learning systems (CLS) theory [10, 11]. CLS theory postulates that intelligent agents require two learning systems: one...
which rapidly acquires new episodic experiences to form short-term memories (encoded by hippocampal networks in vertebrates), and another which generalizes those experiences and store them within existing pool of knowledge as long-term memories in the neocortex. By working in tandem, these systems can weight novel experiences and integrate them effectively with long-term knowledge without damaging past memories [12]. RL techniques integrating ideas from CLS have been implemented and been shown to be effective at a variety of tasks [13–15].

In this work, we show that a ‘dual-memory learner’ (DML) can also be effective at RL tasks on a neuromorphic platform. Applying a highly simplified view of CLS, this architecture uses separate modules to store episodic memories and long-term knowledge. The full DML architecture is implemented in a spiking manner and executed on Intel’s Loihi processor. We demonstrate it solving the classic multi-arm bandit (MAB) problem, as well as more advanced tasks such as navigation through a maze, and the card game blackjack.
To our knowledge, these advanced multi-step problems have not previously been demonstrated being solved solely by a neuromorphic system. We characterize the performance of its current implementation, comment on its characteristics and limitations, and describe improvements which it can undergo in future work.

2. Methods

2.1. Dual-memory learner (DML) framework

Monte Carlo (MC) methods provide well-characterized RL techniques for learning optimal policies via episodic experiences; the agent does not need to be equipped with a full model of how the environment will react to its actions in order to learn. Instead, the agent tracks which states it has entered, the actions it has taken, and once an episode concludes, updates its value estimates given its trajectory through the state space. This provides a simple but effective basis for RL, and we focus on implementing this method in our architecture (though it can also be extended to more modern temporal difference and n-step algorithms) [9].

We propose a DML framework, which is inspired by current theories on information processing in the brain, particularly the so called complimentary learning systems theory [10, 11] to implement an MC learning technique using spiking networks. The proposed DML architecture contains four major sections which process and store the information required to carry out RL on a neuromorphic platform.

The fundamental epoch or ‘step’ in an RL agent modeling a Markov decision process requires several sub-steps: recognizing the agent’s current state, using this information to decide on an appropriate action given the current policy, returning this action to the environment in a meaningful manner, and possibly applying reward signals to update internal value estimates and policy. We define specific modules and/or interactions to address each of these requirements, forming the core structure of the DML and allowing it to be implemented via parallelized and local operations. The four modules we define are the decoder, short-term memory (STM), long-term memory (LTM), and encoder (figure 1).
Figure 4. Illustration of the value circuit (VC). (a) Reward signals have a chance to increase the firing rate of the soft-reset neuron, and punishment signals, a chance to decrease the firing rate. By applying feedback, these probabilities are manipulated to be equal to one another when the firing rate of the soft-reset neuron equals the proportion of positive reinforcement signals received over a memory period (b).

2.2. Architectural implementation

One of the crucial aspects of a neuromorphic system is the question of how information is to be represented, particularly when all information must be encoded in ‘spikes.’ Spikes are a binary all-or-nothing signal which is responsible for much of the information transfer between neurons in the human brain (although non-spiking interactions via glial cells can also play an important role in the brain [16–19]). In this initial implementation, the convention that all information is conveyed via rate-coded spikes is used and thus spike timing information is not used. While rate-coding can be costly compared to other encoding strategies, it is commonly used in neuromorphic solutions due to its easy interpretability and functionality [20, 21]. Using this assumption and following the previously laid-out requirements, we independently demonstrate the operation of individual modules which together implement a rate-coded version of the dual memory learner (RC-DML). With the sole exception of the final encoder module, all modules are implemented entirely using spiking logic which run in the massively parallel ‘Neurocores’ of the Loihi architecture [5].

Spiking networks on Loihi run forward through time in discrete steps. During each execution step, each neuron can fire once. As a consequence of using rate-coding, the RC-DML must run for multiple execution steps to allow spiking statistics to accumulate before taking an action to complete an RL epoch. The appropriate number of execution steps required to carry out an epoch is an important parameter for the network which we investigate experimentally.

2.3. Decoder

In the simplest case, the decoder provides a simple integer value which corresponds to a unique state in the problem. Signals from the environment indicating a change in state are assumed to be sparse in time, and the decoder must take these sparse signals and expand them into a constant internal representation (figure 2). This
Figure 5. Overview of the LTM module. (a) This module retains estimates of the expected value of each state-action pair. (b) An array of VCs maintains these estimates as rate-coded values. (c) These values are adjusted through reward signals routed from the STM.

is done by creating a series of bistable neurons which provide a one-hot representation of the current state (e.g. if an agent in a maze is in the fourth possible position, its fourth neuron in the encoder will be active).

2.4. Short-term memory (STM)
When combined with a similar one-hot representation of the action taken following the current state, the trajectory of the agent through each episode can be built by taking the outer-product of the decoder’s state representation and the consequent action from the encoder (figure 3). Again, bistable neurons are used which are activated once a state-action pair has been traversed, and are reset once an episode is complete (e.g. in a maze, an STM neuron becomes active if the agent entered a location and made a specific movement). In all examples presented here, the end of an episode is indicated by a signal from the environment, optionally accompanied by a positive or negative reward (figure 3(c)).

2.5. Value circuit (VC)
To effectively learn the value of state-action pairs in an RL problem, the RC-DML must have a component which is able to learn and present continuous values. Currently, this is done by maintaining a tabular array of value estimates for each state-action pair. This avoids requiring a function approximator, the training and implementation of which is in itself currently an area of intense study within neuromorphic computing [22, 23].

To represent a single value estimate, a circuit of several neurons is used (figure 4). The value circuit (VC) is a small spiking circuit which provides the LTM and learning capability of the RC-DML. It accomplishes this task by taking an approximate moving average of sparse reward signals and representing this value with a continuous stream of rate-coded spikes.

The VC operates by using feedback to manipulate the probability its firing rate will change when a reward or punishment signal is received. The VC’s output is given by the firing rate of a soft-reset integrate-and-fire (SRIF) neuron with no leakage; this three-compartment neuron produces a regular firing rate when the charge in its memory compartment is held constant. This firing rate is linearly proportional to the amount of charge in the memory compartment versus the integrator’s firing threshold. In order to increase the firing rate, charge is added to the memory compartment through an excitatory synapse, and to decrease it, charge is removed through an inhibitory synapse.

However, if we wish to use this neuron to track the expectation of a reward, charge should not be modified with the arrival of every reward or punishment signal; this is clear if the neuron is already firing tonically or
is quiescent. More generally, by using the spiking output of the SRIF in conjunction with an ‘and’ gate which only fires when spikes are present on all its inputs in a single time-step, punishment signals can be blocked or passed to make modifications to charge which are dependent on how close to the quiescent state the SRIF is (figure 4(a)). To create a similar condition for the reward signals, an inverter is added to the signal path which fires whenever an input is not present. This restricts reward signals, making them less likely to pass the closer the SRIF is to the tonic state. By combining these two boundaries, the probabilities that the SRIF’s firing rate will increase or decrease become equal when the firing rate matches the proportion of reward signals out of reinforcement signals received over a finite period (figure 4(b)).

The convergence of the SRIF is shown formally by equations (1)–(3) where \(+\Delta q\) and \(-\Delta q\) are increases or decreases (respectively) in SRIF charge, \(s\) is the state of the SRIF’s output, \(\hat{r}\) is the SRIF’s firing rate estimating the true parameter of Bernoulli-distributed reward \(r\), and reward/punishment indicates a sample of 1/0 from this distribution.

\[
p(+\Delta q) = p(s = 0|\hat{r}) \cdot p(\text{reward} = 1|r) = (1 - \hat{r}) (r) \quad (1)
\]

\[
p(-\Delta q) = p(s = 1|\hat{r}) \cdot p(\text{punish} = 1|r) = (\hat{r}) (1 - r) \quad (2)
\]

\[
p(+\Delta q) = p(-\Delta q) \quad \text{iff} \quad \hat{r} = r \quad (3)
\]

2.6. Long-term memory (LTM)
An array of VCs represents the expected returns for all state-action pairs following the current policy (figure 5). Agent policy is formed from a simple greedy or \(\varepsilon\)-greedy choice on action value estimates given the current state. At the end of an episode, these value expectations are updated by using the trajectory stored in the STM to route reward signals to their corresponding VCs. These signals then can incrementally adjust each VC as necessary, creating new value estimates and allowing new policies to be derived.

2.7. Encoder
In order to select an action appropriate to the current state, the encoder uses the information from the decoder to filter the output of the LTM (figure 6).

Reading only the outputs of the LTM which correspond to actions possible at the current state, the decoder chooses the action with the highest value (greedy), or optionally, may instead choose a random action with a set probability (\(\varepsilon\)-greedy). Currently, the required argmax and random selection operations are done via the
Figure 7. Results from the RC-DML solution to the MAB problem. (a) The parameters of the Bernoulli distributions used to draw rewards after the pull of an arm. (b) The choices of the RC-DML over 2000 epochs (arm pulls). (c) The mean optimal action (MOA) of the RC-DML bandit, averaged over 100 epochs. (d) The MOA of the bandit as the length of its rate coded representations is varied. Bars indicate standard deviation ($n = 5$). (e) Comparison of the MOA between the RC-DML executing on Loihi and a traditional CPU-based $\epsilon$-greedy algorithm. (f) Further comparisons between the RC-DML and CPU $\epsilon$-greedy algorithms measuring MOA and cumulative reward as $\epsilon$ is varied. Bars indicate standard deviation ($n = 5$).
x86 processor co-integrated on the Loihi chip, though they may be replaced via a winner-take-all (WTA) or noisy WTA circuit for a purely spiking implementation of the DML [24]. Other action selection methods such as softmax-greedy or random choice may be implemented by modifying this module.

2.8. Modular integration

Having demonstrated the individual operation of each DML module, the remaining challenge is to integrate these modules into a system which works in concert to implement the full DML. Additionally, this should be achieved by describing modules at a high level of abstraction, allowing the solution to automatically scale with the problem at hand and preventing it from becoming burdensome to end users who may wish to deploy the program into new scenarios. This remains a challenge in neuromorphic systems where ‘completeness’ is debatable and compilation of arbitrary programs to end platforms may not always be feasible [4].

We maintain high levels of abstraction in our program integration by describing neuronal circuits and hierarchies of circuits in terms of computational graphs. The elements of these graphs are nodes with arbitrary dimensions, linked with stereotyped connectivity patterns and predefined excitability characteristics. These circuits can then be easily scaled to the given problem at hand and compiled down into the individual compartments and synaptic connections required to define an executable program on a platform such as Loihi. The block diagrams representing these graphs are provided in the supplementary materials (https://stacks.iop.org/NCE/1/024003/mmedia).

This high-level organization allows the RC-DML to address a variety of different problems while requiring minimal amounts of code updates and maintaining executability on neuromorphic hardware. The only code changes required to allow the agent to address different problems are routines which update the simulation of the environment and control communication between the agent, environment, and host computer. The full source code for the RC-DML on Loihi and our experiments is openly available (https://github.com/wilkieolin/loihi_rl).

2.9. Power consumption

An important aspect of any neuromorphic solution is to characterize the power consumption which it produces compared to more common implementations [21, 25]. To characterize the performance of the RC-DML executing on Loihi, power estimates on the Loihi system was measured by using energy probe tools included in the NsSDK toolkit.

To provide a baseline CPU comparison, an equivalent MC RL program using conventional (non-spiking) method was implemented in the Julia language for the blackjack task presented below (available in repository). Power estimates for CPU task were estimated by utilizing the Intel SoC Watch energy analysis profiler on a Windows 10-based system with an Intel i7-4710HQ processor (22 nm node). Active power used to calculate the optimal blackjack policy was measured by running the program 5 times and subtracting the system’s baseline power usage (also measured over 5 independent trials).

3. Results

3.1. Multi-arm bandit

As a basic demonstration of its capabilities, we first apply the rate-coded DML (RC-DML) to the MAB problem. While it does not incorporate the concept of ‘state’ into a problem, the MAB is itself a complex problem which addresses many fundamental aspects of RL [9]. In the MAB, a series of ‘arms’ is presented to the agent, each with a hidden true parameter which controls the probability a reward is obtained when that arm is ‘pulled’ [26]. Here, the fundamental learning problem is to find which arm gives the highest reward using the smallest number of interactions with the bandit in order to maximize cumulative reward.

We demonstrate RC-DML using an ε-greedy algorithm to solve the MAB. As the problem is stateless, it is simply indicated to the system that it remains in the same (singular) state after every action. The value estimates learned for each action in this state are then used to estimate the reward for each arm (figure 7(a)). Using an ε-greedy policy, the RC-DML is forced to explore each arm and eventually converges on selecting the correct arm (figures 7(b) and (c)).

As long the length of the period used to rate-code values lies above approximately 40 execution steps, the neuromorphic RC-DML demonstrates learning performance for the MAB which is comparable to that of a traditional, non-spiking ε-greedy algorithm running on a CPU (figures 7(d) and (e)). Except where noted, in all experiments we use an epoch length of 128 execution steps, well above this lower boundary.

The CPU-based algorithm maintains a small performance lead in proportion of optimal actions and the mean average reward over the first 6400 epochs of learning over a variety of ε values (figure 7(f)). This is due to the limited accuracy of value representation in the RC-DML due to its use of rate-coding, which later limits its
performance at blackjack. No such obstacles are present in the CPU-based algorithm which uses 64 bit floating point representations of value.

3.2. Dynamic maze

Navigation is a long-studied problem with many practical applications, such as, e.g., robotic cleaners and self-driving vehicles. We focus on a dynamic maze task to evaluate the RC-DML’s ability to learn navigation patterns. In this dynamic maze, a target location within a rectangular grid of points provides a reward. If the agent can reach this reward within a set number of moves, it is rewarded. During this time, the only information provided to the agent is its current location within the maze. It cannot ‘see’ walls, but infers their presence by detecting its location remains unchanged after attempting to move in a certain direction. If the agent fails to reach the goal within a set number of steps, it is punished and a new episode begins. The multi-state nature of navigation requires the RC-DML to demonstrate that it can accurately incorporate an episodic memory into its learning process. Additionally, the RC-DML here employs a greedy policy combined with a random starting location to force exploration.

Given the consistent nature of rewards, the RC-DML can quickly learn the location of the reward within a 5 by 5 grid and converge on a policy which navigates to it from all starting locations within the time limit of 8 steps (figure 8(a)). When walls are introduced within the maze to block off the reward from 3 sides, the previous navigation policy conflicts with the new constraints imposed by the environment. But as the value estimates within the RC-DML are effectively created by a running average of previous returns, it gradually overcomes this faulty policy as the new experiences gained from the environment update value estimates and allow a better policy to emerge which correctly navigates to the reward (figure 8(b)).
Figure 9. Results from applying the RC-DML to playing blackjack. (a) Optimal policy for blackjack obtained by conventional MC solution on CPU after $2 \times 10^9$ episodes. (b) Blackjack policy found by RC-DML after $1 \times 10^6$ episodes. Much of its strategy is correct (e.g. sticking at high values), but consistent differences from the optimal policy remain. (c) Jensen–Shannon divergence between the RC-DML’s policies and the optimal CPU-calculated policy. The coarse representation of values within the RC-DML limit its performance in this task and prevent it from reaching the optimal policy. (d) The difference in policies between CPU and RC-DML corresponds to states where the difference in value between actions is small and requires fine representation.

Finally, the reward is placed at a new location within the maze. This requires a greater shift in the navigational policy than when the previous reward location is blocked off, and the RC-DML takes a longer time to adjust to this new environment, but once again the optimal policy is reached with experience (figures 8(c) and (d)).

3.3. Blackjack
The card game blackjack contains elements of random chance which makes it challenging to quickly converge to the optimal policy. In the variation of blackjack we implement, the goal is to obtain a hand of cards with values summing as close as possible to 21 without going over. Each pip card [2–9] has a value equivalent to its number, face cards are worth 10 points, and the ace can count as either 1 or 11. To begin play, a card is dealt to both the dealer and the player. From there, the player chooses to either receive a new card from the dealer (‘hit’) or stay with their current sum (‘stick’). The dealer then follows a fixed policy to draw cards until their sum is 17 or greater. If either the player is closer to 21 or the dealer goes over 21 during their turn, the player
wins. If both player and dealer are equally close to 21, the game is a draw. Otherwise, the dealer wins and the player loses.

To solve this problem, the RC-DML again uses a greedy policy combined with exploring starts. The state of the player in blackjack is entirely determined by the current sum of the player’s cards (player sum), whether this sum can be modified by counting an in-hand ace as either 1 or 11 without exceeding 21 (usable ace), and the value of the dealer’s visible card (dealer showing). Cards are drawn from an infinite deck, therefore there is no hidden state within the dealing process to consider.

Although the RC-DML converges onto a policy which is quantifiably better than random chance and does not make obvious mistakes (such as drawing a new card when the player sum is already close to 20), a gap remains between its performance and that of an equivalent MC algorithm on CPU (figures 9(a)–(c)). As previously mentioned, the limitations of rate-coding value estimates are the main source of this performance gap. Compared to the previous navigation problem, learning an optimal blackjack policy requires much more fine-grained representations of value (figure 9(d)).

In a problem such as blackjack (with the largest state-action space of all tested problems at 400), this means much of the energy consumed to produce spikes is wasted, as only 0.5% of estimates being produced are relevant to the current state during each epoch. As a result, while the active power consumption of a Loihi chip running the RC-DML algorithm to solve blackjack is much lower than that of a traditional CPU, the faster execution rate of an equivalent MC program on CPU gives it an advantage in the amount of energy consumed per RL epoch (67 μJ/epoch on CPU vs 1728 μJ/epoch on Loihi when using a reduced 64 execution steps per epoch).

4. Discussion

We have presented a novel framework for solving RL problems using spiking neuromorphic hardware. This method implements modules inspired by the CLS theory which was proposed to explain declarative memory learning in the biological brain and depends on the interaction between hippocampal and cortical networks. The method was successfully applied to three classic RL problems—MAB, dynamic maze, and blackjack—using Intel’s Loihi neuromorphic processor. We found that Loihi based implementation presents similar performance to the CPU based algorithms unless high precision was required to successfully learn a policy. While the active power consumption of a Loihi chip running the algorithm was much lower than that of a traditional CPU, for more complex problems, such as blackjack, the faster execution rate of an equivalent MC program on a CPU gave it an advantage in the overall energy consumption. The last limitation was found to be a result of rate-coding implementation of the information processing, that we used here due to its easy interpretability and functionality.

4.1. Reinforcement learning in machine learning solutions

Since its formal emergence in the mid-20th century, RL has grown to provide a variety of techniques which can provably converge on optimal solutions for complex problems [9]. With the addition of modern deep-learning techniques, RL has surpassed human performance on a variety of problems, including the classic games Chess et al. [27]. Due to its ability to conquer these complex scenarios, RL has many possible applications in real-world problems including autonomous driving, process control, and interpreting biological data [28–30]. However, deploying RL in these and other edge-case situations requires that it be implemented in an efficient manner [1]. The desire for low-power implementations of RL, combined with the theory that neural circuits carry out some form of RL, has motivated several previous efforts toward implementing RL on neuromorphic platforms [9, 31–33].

The fundamental problem which RL solves is providing an ‘agent’ in an environment a way of updating its internal models to choose an action at each state of the problem which leads to a maximized cumulative reward, or ‘return.’ Generally, this is done by iteratively improving the agent’s value estimates, which are then applied to form a new policy to guide actions. Value estimates following the new policy are updated, and the process repeats. As long as this process satisfies the Bellman optimality principle, this process is guaranteed to converge to an optimal policy [9].

Implementing these mechanics on a neuromorphic platform gives rise to a number of challenges, such as how to represent and update values, how to update these values using local information, and how this program can be expressed in a manner which may be implemented on multiple architectures despite their underlying differences [2, 4, 22].

To address these challenges, we created a flexible program carrying out an RL strategy described by a high-level computational graph. In order to create this program, we focused on the requirements of an RL program and how they can be satisfied by a biologically-inspired structure. Once defined, we implemented this program
on a neuromorphic platform in a modular manner, allowing it to address different problems with minimal changes from the user.

One challenge for the current architecture stems from the fact that it currently implements only a MC RL technique which requires a full episodic experience to be completed. In complex problems, episodes may be either long or unclearly defined, and the MC technique of delaying value updates until an episode is complete can both require large amounts of memory and slow the learning process. We believe, however, that proposed conceptual framework can be extended to address this limitation by changing the nature of the STM/LTM interaction. By storing limited traces of two or more states and sampling expected rewards from the LTM as well as true values from the environment, the LTM can be continually updated using temporal-difference or n-step methods [9].

4.2. Learning in biological brain and complimentary learning systems theory

The mammalian brain contains analogues to the functions required for RL methods. Dedicated areas of the brain focus on maintaining robust internal representations of state, including physical location and proprioception [34, 35]. It is believed that the hippocampus can fuse these complex representations of state over time, allowing animals to ‘record’ the paths they have taken in physical environment and play them back [36, 37]. Short-term memories rapidly formed in hippocampus can then be used to build more complex representations and strategies as they are integrated into the larger, LTM provided by the neocortex. CLS theory [10, 11] suggests that these two areas provide complimentary functions to one another: the hippocampus provides a highly-plastic memory to learn from new experiences, and the neocortex, by utilizing spontaneous memory replay [12], integrates this new information into generalizable representations which become highly stable over time. This theory has provided a rich set of inspirations for novel techniques in RL which can increase its sample-efficiency while avoiding catastrophic forgetting [15].

In this work, we implemented modular processing sections inspired by this biological viewpoint. Working in concert, these modules cooperatively provide the processing required to implement an MC-based RL method. By changing the role carried out by each module and the interaction(s) between them, this architecture may be adapted to new tasks and learning methods.

The decoder’s role is to interpret and preserve information from the environment. Viewing sensory information from the environment as a coded signal, the decoder’s task is to extract everything from this signal relevant to the current state and store this internally for other modules to use (e.g. given an image of a chess board, evaluate the placement of the pieces and represent this state information in a manner meaningful to the downstream modules).

The STM is responsible for storing the agent’s episodic memory. Given internal representations state, action, and reward the agent experiences during an episode, the STM must store this information and allow it to be ‘replayed’ when the episode has been completed and can be used to update value estimates. Consequently, the STM utilizes a highly plastic memory which updates with every action the agent takes.

One of the central pieces of any RL agent is its value estimator, which here must provide the return expected given the current state, next action, and policy. This information must be constructed via cumulative experiences and/or internal bootstrapping, with each additional experience building upon previous knowledge rather than overwriting it. This requires the value estimator to have a stable, LTM. This memory is gradually updated by applying the state trajectory stored in the STM, combined with a reward, to update current value estimates. Ultimately, the value estimates stored in the LTM are responsible for constructing the agent’s current policy, controlling its decisions.

The encoder carries out the final sub-step in an RL epoch, applying the current state and policy to choose an action. Additionally, this action may need to be transformed from an internal state into one which can act on the environment (e.g. the intent to move a piece in chess must be translated into the physical act of picking up and moving an object). These tasks are carried out by the encoder, closing the loop of the RL process.

Having defined these modules and how they act in concert to perform RL, the challenge was to translate them from abstract concepts into realizable implementations which can be implemented on a real neuromorphic processor. In this work, we focused on demonstrating the architecture with a simple proof-of-concept implementation which can be easily interpreted. However, more advanced representations and other advances in neuromorphic computation can be integrated into the model in the future, and we posit this could lead to large gains in efficiency over the current implementation.

4.3. Power consumption

One of the key goals of neuromorphic hardware is to increase the efficiency of learning and inference in artificial systems [2]. However, the current implantation of the DML makes several trade-offs which reduce its
energy efficiency. First, information is represented through rate-coding; this requires that each step of inference in a decision-making process be run over several time steps in order to collect statistics on current spike rates [38]. Second, a dedicated spiking neural circuit exists in the LTM for every state-action value which must be estimated. These circuits run continually and in parallel, even when their information is not needed.

4.4. Current issues and future directions
Two key issues of the RC-DML (its uncompetitive power consumption and poor scalability) stem from the same underlying source: the rate-coded and tabular implementation of value estimates in the LTM. We argue that replacing this block with a different representation of information calculated through a function approximator which stores information passively in synapses would succeed in making the DML competitive with traditional CPU implementations, both in terms of learning capability and energy efficiency.

For instance, vector-symbolic architectures (VSAs) offer an alternative basis of representing information which could be leveraged within the DML [39]. Binary spatter codes in particular provide a clear method toward replacing the rate-coding of information [40]. This would both increase the number of values a single population of neurons can represent and reduce the amount of time required to produce a representation to a single time-step. Potentially, this single change could allow the DML to outperform the CPU in per-epoch energy consumption, as running the current RC-DML architecture reduces its power consumption to 27 μJ/epoch when only a single computation step is executed per RL epoch.

Storing information in a vector-symbolic manner may also allow for the unification of learning and planning within the LTM. VSAs offer the capability to encode advanced data structures within vectors, potentially allowing for representations of both values and subsequent states to be created [41]. These combined representations could then be retrieved and decoded to provide the basis of planning capabilities.

A larger challenge facing neuromorphic implementations of the DML is an efficient and effective method for training function approximators such as deep neural networks. This advance was a key requirement for recent progress in RL [9], and equivalents within neuromorphic platforms must exist to enable state-of-the-art RL techniques on these platforms. However, advances have been made on this front and should be evaluated to be incorporated into future revisions of the DML on neuromorphic platforms which support them [22, 23, 42, 43]. Applying these learning methods to create function approximators which store weights in synapses (as opposed to sustained spiking activity) would allow the architecture to scale to more complex problems with larger state-spaces.

5. Conclusion
RL provides unique learning capabilities and its development has provided many landmark successes over the past decade. Therefore, it is crucial for neuromorphic systems to show that they are capable of RL techniques and can demonstrate advantages for these techniques over traditional hardware. In this work, we have demonstrated a flexible architecture for RL on neuromorphic hardware which was implemented and fully executed on the Intel Loihi platform. This rate-coded dual memory learner (RC-DML) was successfully able to learn policies to maximize reward received from a MAB, navigate through a changing maze, and play the card game blackjack. But while this shows that a neuromorphic architecture is currently capable of RL techniques, the current implementation’s rate-coded and tabular approach to value representation makes it uncompetitive with traditional technologies. However, the further advances from research into neuromorphic systems (such as value representation through VSAs and spike-based deep learning) can potentially overcome this obstacle to create a system which can match the performance of traditional approaches with greater energy efficiency.

Acknowledgments
We would like to thank the members of the Intel Neuromorphic Research Lab for their support during our software development process.

This work was supported by Intel (00018020-001) and the Lifelong Learning Machines program from DARPA/MTO (HR0011-18-2-0021).

Data availability statement
The data that support the findings of this study are openly available at the following URL/DOI: https://github.com/wilkieolin/loihi_rl.
References

[1] Lin S-C, Zhang Y, Hsu C-H, Skach M, Haque M E, Tang I and Mars J 2018 The architectural implications of autonomous driving ACM Sigdton Not. 53 751–66
[2] Schuman C D, Potok T E, Patton R M, Birdwell J D, Dean M E, Rose G S and Planck I S 2017 A survey of neuromorphic computing and neural networks in hardware (arXiv:1705.06965)
[3] Mead C 1990 Neuromorphic electronic systems Proc. IEEE 78 1629–36
[4] Zhang Y et al 2020 A system hierarchy for brain-inspired computing Nature 586 378–84
[5] Davies M et al 2018 Loihi: a neuromorphic manycore processor with on-chip learning IEEE Micro 38 82–99
[6] Kenji D 1999 What are the computations of the cerebellum, the basal ganglia and the cerebral cortex? Neural Netw. 12 961–74
[7] Kenji D 2000 Reinforcement learning in continuous time and space Neural Comput. 12 219–45
[8] Glümer C P 2011 Understanding dopamine and reinforcement learning: the dopamine reward prediction error hypothesis Proc. Natl. Acad. Sci. 108 17569
[9] Sutton R S and Barto A G 2018 Reinforcement Learning, an Introduction 2nd edn (Cambridge, MA: MIT Press) p 552 Available from: http://incompleteideas.net/book/the-book.html
[10] McClelland J L, McNaughton B L and O’Reilly R C 1995 Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory Psychol. Rev. 102 419–57
[11] Kumaran D, Hassabis D and McClelland J L 2016 What learning systems do intelligent agents need? Complementary learning systems theory updated Trends Cognit. Sci. 20 512–24
[12] Diekelmann S and Born J 2010 The memory function of sleep Nat. Rev. Neurosci. 11 114–26
[13] Schaul T, Quan J, Antonoglou I and Silver D 2016 Prioritized experience replay 4th Int. Conf. Learn Represent ICLR 2016–Conf. Track Proc. pp 1–21
[14] Blakeman S and Mareschal D 2020 A complementary learning systems approach to temporal difference learning Neural Netw. 122 218–30
[15] Botvinick M, Ritter S, Wang J X, Karth-Nelson Z, Blundell C and Hassabis D 2019 Reinforcement learning, fast and slow Trends Cognit. Sci. 23 408–22
[16] Gerstner W and Kistler W M 2002 Spiking Neuron Models: Single Neurons, Populations, Plasticity (Cambridge: Cambridge University Press)
[17] Haydon P G 2001 Glia: listening and talking to the synapse Nat. Rev. Neurosci. 2 185–93
[18] Pannasch U, Vargová L, Reingruber J, Ezan P, Holcman D, Giaume C, Sykova E and Rouach N 2011 Astroglial networks scale synaptic activity and plasticity Proc. Natl. Acad. Sci. 108 8467–72
[19] von Bartheld C S, Barney J and Herculano-Houzel S 2016 The search for true numbers of neurons and glial cells in the human brain: a review of 150 years of cell counting J. Comp. Neurol. 524 3865–95
[20] Dupeyroux J 2021 A toolbox for neuromorphic sensing in robotics (arXiv:2103.02751)
[21] Davidson S and Furber S B 2021 Comparison of artificial and spiking neural networks on digital hardware Front. Neurosci. 15 651141
[22] Tavanei A, Ghodrati M, Kheradpisheh S R, Masquelier T and Maida A 2019 Deep learning in spiking neural networks Neural Netw. 111 47–63
[23] Nefci E O, Mostafa H and Zenke F 2019 Surrogate gradient learning in spiking neural networks (arXiv:1901.09948)
[24] Lynch N, Musco C and Parter M 2019 Winner-take-all computation in spiking neural networks (arXiv:1904.12591)
[25] Davies M, Wild A, Orchard G, Sandamirskaya Y, Guerra G A F, Joshi P, Planck P and Risbud S R 2021 Advancing neuromorphic computing and neural networks in hardware (arXiv:1705.06965)
[26] Kaufmann E, Capp´e O and Garivier A 2012 On Bayesian upper confidence bounds for bandit problems Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics 22 592–600
[27] Silver D et al 2018 A general reinforcement learning algorithm that masters chess, Shogi, and go through self-play Science 362 1140–4
[28] Ravi Kiran B, Sobh I, Talpaert V, Mannion P, Al Sallab A A, Yogamani S and Perez P 2020 Deep reinforcement learning for autonomous driving: a survey (arXiv:2002.00444)
[29] Hafner R and Riedmiller M 2011 Reinforcement learning in feedback control: challenges and benchmarks from technical process control Mach. Learn. 84 137–69
[30] Mahmoud M, Shamim Kaiser M, Hussain A and Vassanelli S Applications of deep learning and reinforcement learning to biological data 2017 IEEE Trans. Neural Netw. Learn. Syst. 29 2063–79
[31] Friedmann S, Préaux N, Schremel J, Gerstner W and Meier K 2013 Reward-based learning under hardware constraints—using a RISC processor embedded in a neuromorphic substrate Front. Neurosci. 7 160
[32] Amravati A, Nasir S B, Thangadurai S, Yoon I and Raychowdhury A 2018 Accelerator with stochastic synapses and embedded 2018 IEEE Int. Solid-State Circuits Conf. pp 124–6
[33] Walter F, Rührmair F and Knoll A 2015 Neuromorphic implementations of neurobiological learning algorithms for spiking neural networks Neural Netw. 72 152–67
[34] Solstad T, Moser E I and Einevoll G T 2006 From grid cells to place cells: a mathematical model Hippocampus 16 1026–31
[35] Moser M-B, Bowland D C and Moser E I 2015 Place cells, grid cells, and memory Cold Spring Harbor Perspect. Biol. 7 a021808
[36] Buzsáki G 2019 The Brain from Inside Out (Oxford: Oxford University Press)
[37] Denoelis E L, Gillespie A K, Coulter M E, Kay K, Sosa M, Chung J E and Frank L M 2020 Hippocampal replay of experience at real-world speeds (accessed 08 01 2021) bioRxiv preprint Available from: https://doi.org/10.1101/2020.10.20.347708
[38] Rueckauer B, Lungu I A, Hu Y, Pfeiffer M and Liu S C 2017 Conversion of continuous-valued deep networks to efficient event-driven networks for image classification Front. Neurosci. 11 682
[39] Neubert P, Schubert S and Protzel P 2019 An introduction to hyperdimensional computing for robotics Künstl. Intell. 33 319–30
[40] Schlegel K, Neubert P and Protzel P 2020 A comparison of vector symbolic architectures (arXiv:2001.11797)
[41] Frady E P, Kent S, Olshausen B A and Sommer F T 2020 Resonator networks for factoring distributed representations of data structures (arXiv:2007.03748)

[42] Bellec G, Scherr F, Hajek E, Salaj D, Subramoney A, Legenstein R and Maass W 2019 Eligibility traces provide a data-inspired alternative to backpropagation through time Available from: https://openreview.net/forum?id=SkxJ4QKlS

[43] Stewart K, Orchard G, Shrestha S B and Neftci E 2020 Online few-shot Gesture learning on a Neuromorphic processor (arXiv:2008.01151)